Fall 2007

A real time operating system based test-bed for autonomous vehicle navigation

Pravin Dhake

Department:

A REAL TIME OPERATING SYSTEM BASED TEST-BED FOR AUTONOMOUS VEHICLE NAVIGATION

By

PRAVIN DHAKE

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

2007

_____	_____
Dr. Levent Acar, Advisor	Dr. Jagannath Sarangapani

_____

Dr. Ted McCracken

ABSTRACT

The experiments and research performed on Autonomous Vehicles and Navigation Systems have gained increasing interest over the last few years. Various intelligence algorithms are being tested to generate the optimum paths for the Autonomous Navigation Systems and to provide the necessary intelligence to those systems depending upon the application.

Research and experiments on these Autonomous Navigation Schemes and Algorithms need an efficient test-bed for objective performance analysis. These algorithms often require sensor inputs from the systems such as the speed and steering sensors to apply feedback control action. An efficient test-bed provides status of all sensors and records of all previous sensor values is very desirable.

This work involves developing for such a test-bed to support research on Autonomous Navigation schemes and Algorithms involved in these applications. Different approaches are analyzed and an optimum approach to design test-bed is implemented.

ACKNOWLEDGEMENTS

The author wishes to thank his advisor, Dr. Levent Acar, for his guidance and continued advice throughout the research and preparation of this thesis. The author would like to express his gratitude to Mr. Tim Media (MIT) for his suggestions without which this thesis would not have been possible and to Dr. Jagannath Sarangapani and Dr. Ted McCracken for serving on his committee.

The author would like to thank his parents and his friends for their continued encouragement and blessing toward achieving his goal. The author would also like to thank his friends, especially, Shivakar Vulli and Gerard Sequiera, for the numerous discussions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Autonomous vehicle algorithms often depend upon the current and the past states of the vehicle to decide the next control step. Thus, sensor information from data acquisition hardware and software within a specific guaranteed time becomes imperative in the implementation of these Autonomous Vehicle Control Algorithms. Some of the previous projects on Autonomous vehicle Navigation schemes are given in [4-10].

Some of the challenges faced in the design of Autonomous Navigating Vehicles include developing an efficient data acquisition system to get instant sensor and actuator access without missing any sensor update or actuator control step and making the computing system performance predictable.

This work identifies these challenges and presents a Real Time approach in developing a test-bed for Autonomous Vehicles Algorithms testing using Real Time Application Interface (RTAI), [14].

The test-bed consists of a 1/16 scaled trailer truck, a single board computer, data acquisition hardware, and a real time data acquisition and control software library. The real time system implementation makes possible the prediction of the system response in terms of worst-case latency period and guarantees that any real time execution event is never skipped.

The user of the test-bed implements the control algorithm for the autonomous vehicle control application using the test-bed hardware model presented in [10] and develops the application in real time using the application programming interface (API) provided by the test-bed software library.

In recent years, extensive research and development has been conducted on the design real time systems and different approaches. A detailed analysis of various approaches for designing real time systems was conducted and an approach was implemented considering the system requirements, budget considerations and technical support available.

2. TEST-BED DESIGN METHODOLOGY

The initial challenge is to decide on the necessary variables or state to model and control the test-bed. Providing a software library, that contains all the user-required functions in form of an "Application Programming Interface (API)" callable by the user is the most desirable solution. Providing a software library would make possible listing of all the functions with self explanatory names for obtaining the sensor and actuator access and making them available to the user by simply calling the function names.

The aim would be to facilitate building up a discrete time control system wherein the state variables in form of sensor updates would be provided at sampling instants of one millisecond and the actuator control functions would be provided by the software library.

The next challenge is to decide the sensor reading update strategies. Since there are various methods that could be implemented, they were analyzed for efficiency.

One approach is to develop functions to get sensor update. In the approach the user can access any of sensor update in the control algorithm by calling the respective function. However, this approach would pose many design problems. The main problem would be getting instant access to any state variable. Calling of a function would consume some amount of CPU cycles in getting the sensor update. Problems would also be faced because of scheduling limitations of the operating system, in this case, the non-preemptive embedded Linux kernel. This approach will not be able to guarantee the timing requirements of the control process and hence would present a non-deterministic system. In this case, there is no guarantee that the state variable accessed is valid for that instant.

The limitations of the first approach make possible identification of the system requirements for this project. System requirements are in the form of implementing a deterministic operating system of the test-bed single board computer that can support high frequency applications. In addition, the system should be able to support concurrent running tasks at a high frequency, thus allowing scope for designing a high-speed data acquisition and control system. A software design method that would guarantee instant

real time state variable access was required. To satisfy such requirement, an approach based on writing sensor value update functions in form of higher priority real time concurrent threads and developing the application using Real Time Operating System (RTOS) is to be provided. The threads are to update the sensor values in real time periodically at one millisecond sampling rate. The threads are update the sensor values and then would sleep for one millisecond before making the next update. The CPU would be available for other computations during this interval.

The threads would preempt the existing task and use CPU during time of their updates. Thus, the sensor updates are take place and be available instantly under any load condition on CPU. At any point of time, previous five updates of the sensor readings should always be available and be accessed by simple function calls.

To achieve the desired goals, a real time kernel "Real Time Application Interface (RTAI)" was embedded into the Test-bed CPU. As a result, the sensor updates are guaranteed not to be missed under any load. The application is developed using C and POSIX threads implementation and makes maximum utilization of the RTAI API. The test-bed application contains functions for accessing sensor values at any instant and accessing sensor values of previous five instants.

In this work, the sensor values are updated to arrays, that hold up to five previous sensor values, and functions were developed to access any of the array elements. Functions for actuator control such as starting, stopping, and changing the speed of the drive motor and steering motor have been developed. Emergency functions to stop the vehicle have been developed along with miscellaneous functions to change the PID filter parameters of the motor controller card for the control algorithm optimization.

The application is compiled and developed as a static library. The user only needs to develop the specific control algorithm using the API provided by the library.

An autonomous vehicle system can be modelled as a discrete-time control-system wherein the real time sensor values generate the state variables and the actuator values become outputs of the control algorithm. The test-bed hardware and software are designed be used to implement an autonomous vehicle navigation system as showned in Figure 2.1.

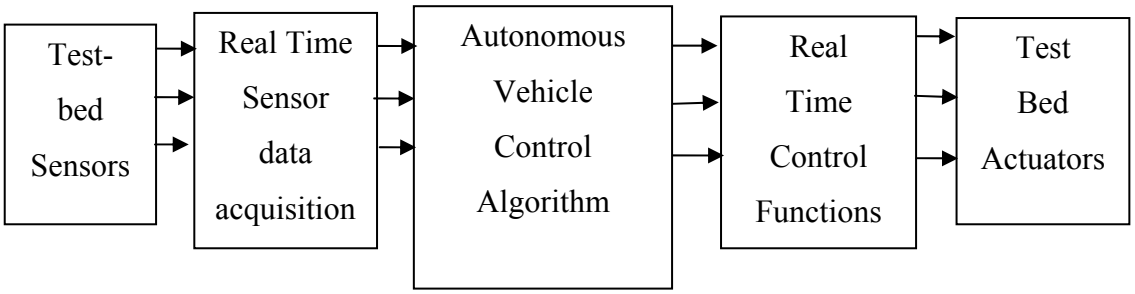

Figure 2.1: System architecture

Numerous intelligent control methods such as Neural Networks ([4], [6]-[8]) or other methods as in [1], [11], can be implemented easily using the developed software library.

3. TEST-BED HARDWARE ARCHITECTURE

3.1. THE TEST-BED CPU

The test-bed consists of an Arcom Control system PC104 based single board computer (SBC) embedded with a Real Time Application Interface (RTAI) based real time Linux kernel. The details of the single board computer configuration is detailed in Table 3.1.

Table 3.1: Test- bed single board computer configuration

Component	Configuration
Processor	AMD Geode TM GX533@1.1W 400MHz clock low-power 32-bit processor
Primary Memory	512MB un-buffered 64-bit 2.5V DDR SDRAM single 200-pin SODIMM
Secondary Memory	32MB Spansion Mirror Bit Flash Data light FlashFX® FlashFX® Pro Flash filing system
Cache	32K L1 write-back cache 16K instruction 16K data
Operating System	Real Time Linux kernel using RTAI

3.2. THE TEST-BED DATA ACQUISITION AND CONTROL HARDWARE

Data acquisition and control are performed by the Arcom AIM104-Multi I/O [19] card and Mesa Electronics 4I27 Motor controller cards, [16]. The AIM104-Multi I/O and Mesa Electronics 4I27 Motor controller cards are PC104 based add on cards that are connected to the test-bed PC104 based single board computer.

3.2.1 AIM104 Multi I/O Card. AIM104 Multi I/O is an 8-bit PC104 module providing 8 opto-isolated digital inputs, 2 analogue outputs (Voltage or Current Loop) for Digital to Analog conversion (DAC) operation and 16 single-ended or eight differential analogue inputs for Analog to Digital Conversion (ADC) operations, [19].

The DAC has channel update time of 320 microseconds per channel, and ADC has channel update time of 500 microseconds per channel.

3.2.1.1 ADC block. The ADC conversion takes place across the channels. The ADC card polls each channel every 500 microseconds. The SBC is pre-loaded with ADC driver software and the data across each channel is accessed as explained in [21].

The standard analogue input range for the AIM104 Multi I/O is ±5V. Signals with greater range than this, such as ±10V, need to be buffered with a simple amplifier circuit. Figure 3.1 explains the input circuitry for the ADC operation in the ±10V range.

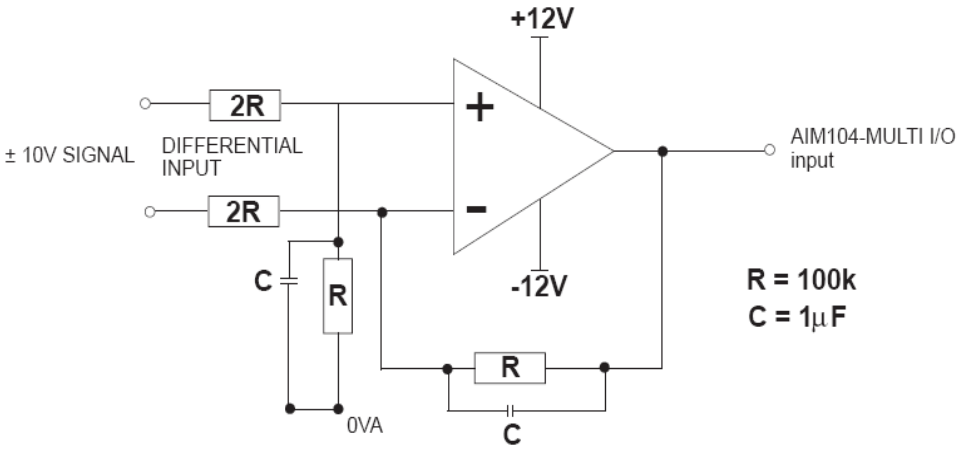

Figure 3.1: ADC input circuitry for ±10V input

3.2.1.2 DAC operation. For the DAC operation, channel numbers are referenced using the corresponding bit numbers in the base address and the status of the digital inputs are read from the base address. Figure 3.2 shows the input configuration.

The DAC conversion takes place across the channels. The DAC card polls each channel every 320 microseconds. The SBC is pre loaded with the DAC driver software and the data across each channel is accessed as explained in [21].

DAC data is written to DAC lower byte and the DAC higher byte registers in accordance with the I/O map. Bit numbers 4 to 7 of the higher byte designate the DAC channel number. A value of '0' in this position writes the data to DAC channel 0.

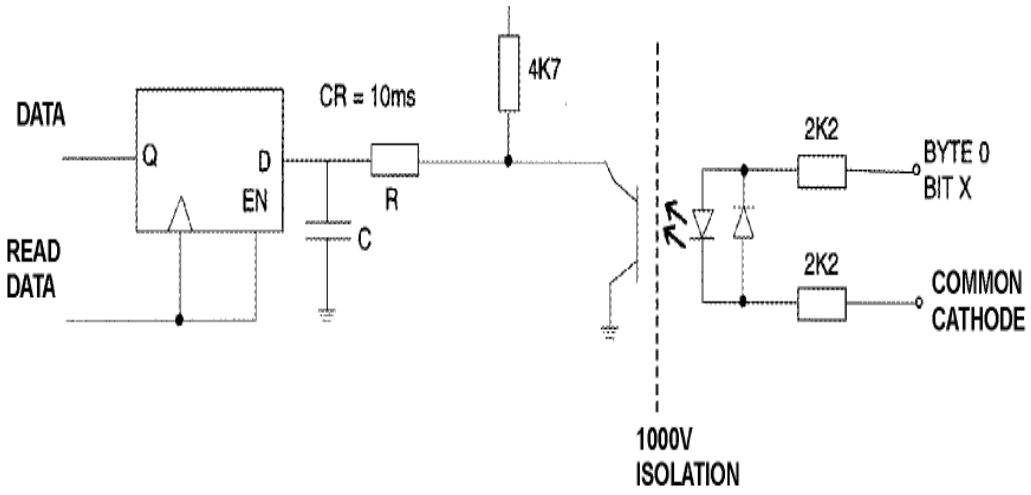

Figure 3.2: Input configuration of DAC

3.2.2 4I27 Motor Controller Card. The 4I27 is a LM629 based 2-axis DC servo motor control system implemented on a stackable PC104 bus card [16]. The 4I27 can be used to operate in velocity mode as well as position mode and uses a digital PID filter to set control parameters for optimization. The 4I27 card provides ability to make changes to Velocity, Target position and control parameters during motion.

The LM629 used on the 4I27 are high performance digital processors specifically designed for motion control. The LM629 are capable of executing a ramp-up, slew, and

ramp-down motion sequence. The LM629 execute these sequences without host processor intervention. The interface software for the 4I27 motor controller card included in the test-bed library, also includes functions to communicate and utilize the sequencing options provided by the 4I27 motor controller card.

Host interrupts can be generated at end of motion, position breakpoints, index pulse, or in response to various error conditions. Interrupts are or'ed on the 4I27 card, so that only one system interrupt is used. The IRQ line used can be software selected from any of the 11 available AT bus interrupts. The test-bed library provides a function for the same operation. 4I27 uses a digital PID filter to set loop feedback parameters for stability and optimum performance. Velocity, target position and filter parameters may be changed during motion. In addition, the clock speed of the LM629 can be lowered to accommodate large motors that require lower PWM chopping frequencies.

The test-bed library provides functions to change the PID filter parameters and also to change the velocity, target position and filter parameters.

Figure 3.3 shows the hardware architecture of LM629 processors.

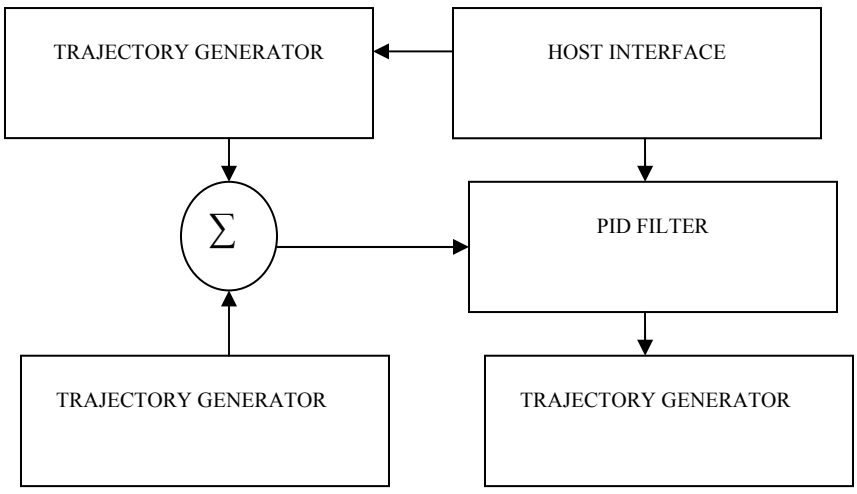

Figure 3.3: LM629 hardware architecture

3.3. THE TEST-BED VEHICLE

The test-bed vehicle is 1/16 scaled trailer truck. The truck is embedded with sensors and actuators, that are accessed and actuated, respectively, by the embedded software. The sensors and actuators are interfaced with the data acquisition and control cards.

The trailer truck test-bed hardware has been modeled and the physical model is described comprehensively in [10] for the user to design a controller. In [10], the test-bed hardware model is represented based on physical system along with the model accuracy results.

For example, the angular velocity of the test-bed trailer cab in terms of steering angle δ is

$$dca * \theta c * \cos\delta = Vc \sin\delta \qquad (1)$$

Where, d_{ca} is the distance between the drive tire and the steering tire, θ_c is the angular velocity and V_C is the drive tire velocity. The steering angle and the drive tire velocity values are provided in real time by the test-bed software library sensor value threads.

The real time software provides the advantage of making theses sensor values always available in real time. The data acquisition software is in real time with the data acquisition threads having the highest priority. The real time implementation ensures that no corrupted or wrong sensor value is in use at any point by the users control algorithm.

Equations governing the trailer truck test-bed system parameters are provided in [10]. The test-bed hardware model gives extensive information of the rules governing the relationship between truck movements and the demands placed on truck motor [10].

In [4] the test-bed hardware model has been successfully utilized to implement path planning control algorithm using neural network. Thus, the model facilitates the user to develop the control algorithm and the test-bed software library API facilitates implementing the control algorithm in software. A detailed documentation of the API is presented in Section 5.

Architecture for control of the autonomous vehicle using the test-bed software library API and the control algorithm itself is presented in detail in Section 4.

Figure 3.4 shows the picture of the test-bed vehicle whose model is presented in [10]. Figure 3.5 shows the test-bed vehicle dimensions. In the figure, the point C represents the drive tires and the point A represents the point of steering angle control tires. Points P and D are the pivot points on vehicle cab and the trailer respectively. The equations for the velocity and acceleration at these points are described and explained in [10].

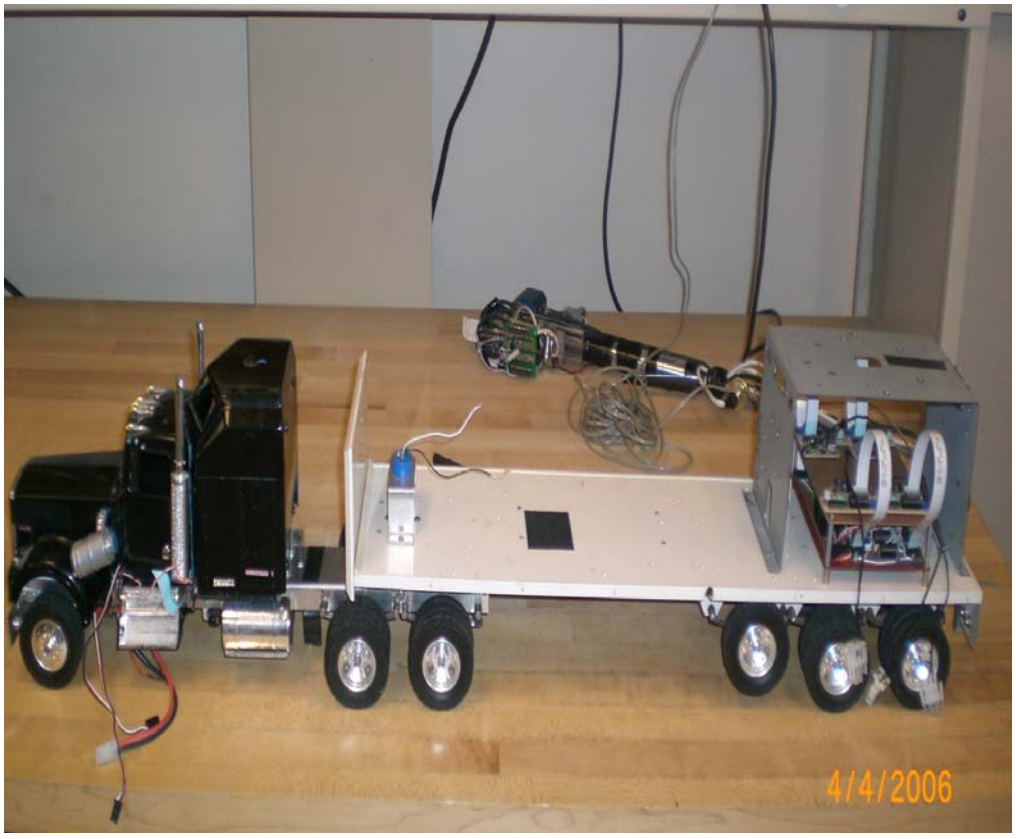

Figure 3.4: Test-bed vehicle

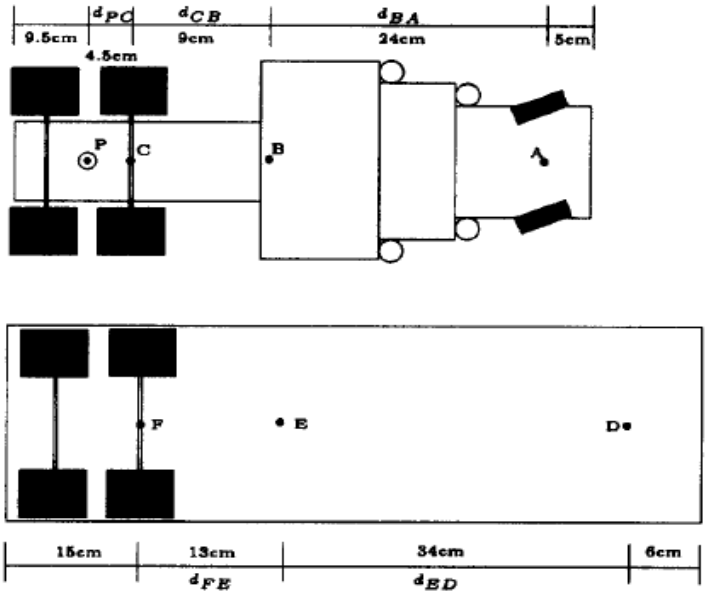

Figure 3.5: Test-bed truck dimensions

3.4. THE TEST-BED SENSORS

3.4.1 Accelerometer. The test-bed features a Dimension Engineering DE-ACCM3D 3D analogue accelerometer [20]. The accelerometer measures acceleration with a minimum full-scale range of ±3 g. It can measure the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion, shock, or vibration. It can enable Vehicle Acceleration logging and can be used as an important state variable in motion control system for the vehicle. The voltage outputs on the accelerometer correspond to acceleration being experienced in the X, Y and Z directions.

The output is ratio metric, so the output sensitivity (in mV/g) will depend on the supply voltage. The accelerometer is powered by a 3.3 v onboard source resulting in range of 0v to 3.3 v. The accelerometer block diagram is given in Figure 3.6 and Figure 3.7 shows the picture of the accelerometer.

Xout, yout and zout are the voltage outputs of the accelerometerthat are calibrated as the accelerometer and slope measurements along x, y and z directions, respectively.

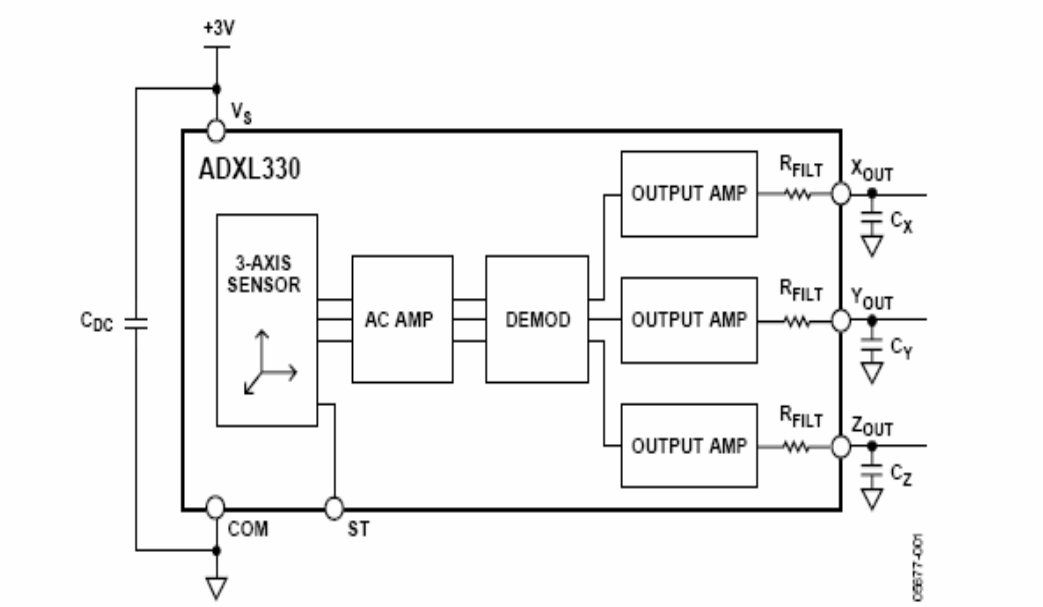

Figure 3.6: Functional block diagram of accelerometer

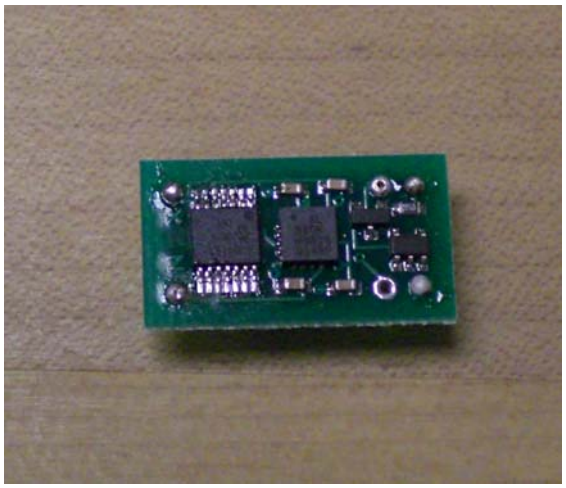

Figure 3.7: Test-bed vehicle accelerometer

3.4.1.1 Theory of operation. The accelerometer contains a micro machined sensor and signal conditioning circuitry to implement open-loop acceleration measurement architecture. The output signals are analog voltages that are proportional to acceleration. The accelerometer can measure the static acceleration of gravity in tilt sensing applications as well as dynamic acceleration resulting from motion, shock, or vibration.

The accelerometer is made up of a polysilicon surface structure suspended over silicon wafer by means of polysilicon springs. A differential capacitor that consists of independent fixed plates and plates that are attached to the moving structure measures the deflection of the structure. 180° out-of-phase square waves are used to drive the fixed plates. Deflection occurs due to the acceleration that deflects the moving structure and creating an unbalance in the differential capacitor resulting in sensor output whose amplitude is proportional to acceleration. The magnitude and the direction of the acceleration are determined by phase sensitive demodulation techniques and are showed in Figure 3.8.

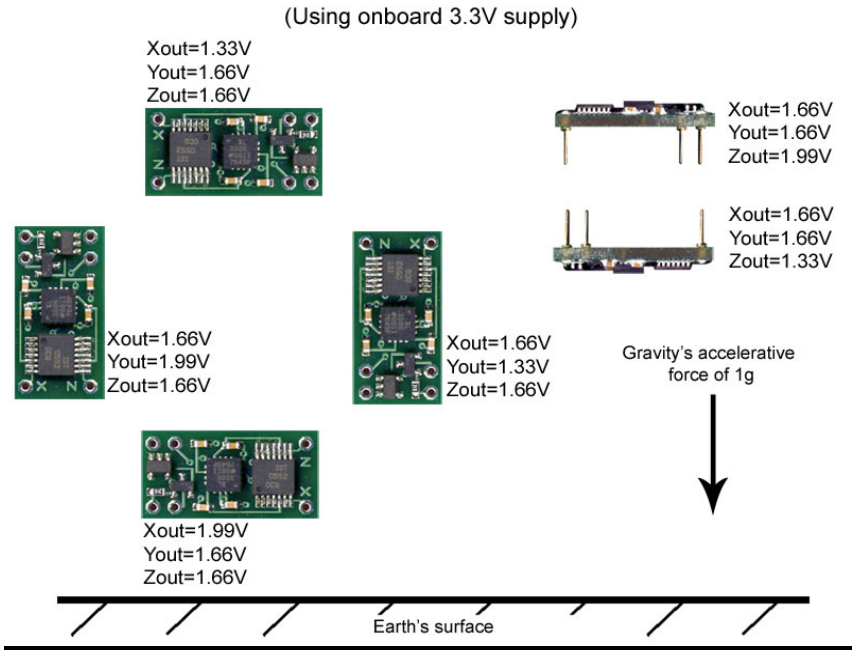

Figure 3.8: Accelerometer magnitude and direction

3.4.1.2 Interfacing with onboard ADC. The accelerometer output is ratio metric, so the output sensitivity (in mV/g) will depend on the supply voltage. The accelerometer is powered by a 3.3 v onboard source. Hence, the output being ratio metric will vary between 0 v to 3.3 v.

The ADC can measure a voltage range from 0v to +5v. However, connecting this voltage straight to A/D input generates low-resolution conversion. There are 4096 possible digital values that the 12-bit ADC can produce, but connecting the accelerometer output directly only utilizes $(3.3/(5-(-5))*4096)=1352$ of the possible 4096 values. Therefore, the solution is to scale the 0 to 3.3 v DC to 0 to 5v DC which is done using DC voltage scaling circuit as shown in Figure 3.9.

The scaling circuit amplifies the input range from 0 to 3.3 v DC to 0 to 5v DC using LM358 operational amplifier. The designing and component selection of the circuit is elaborated in detail.

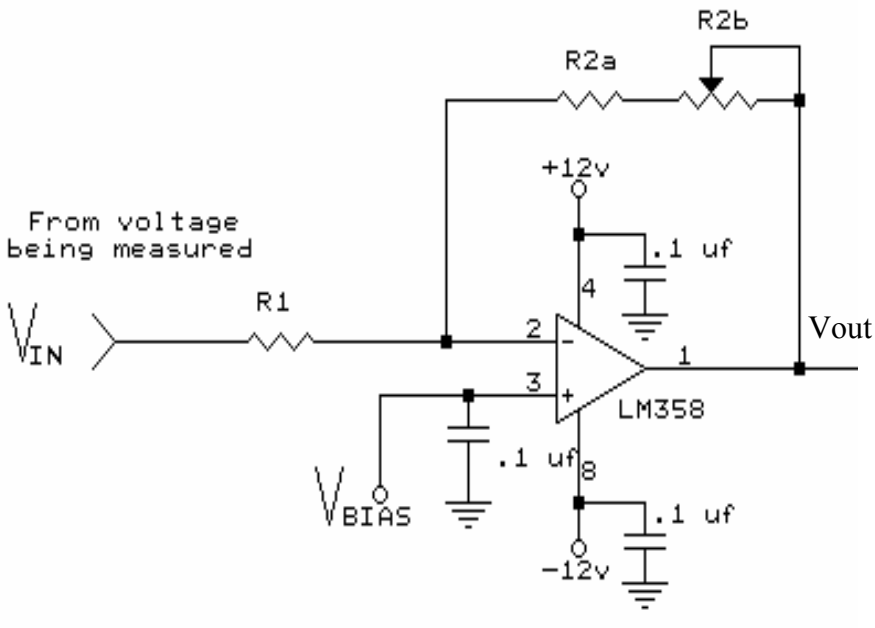

Figure 3.9: DC voltage scaling circuit

Initially, during the design of this signal conditioning circuit, the gain Av is determined as

$$Av = abs(OutputVoltageRange / InputVoltageRange) \quad (2)$$

$$Av = abs((Voh - Vol)/(Vih - Vil)) \quad (3)$$

A_V = DC voltage gain.

V_{ih} = High end of input voltage range.

V_{il} = Low end of input voltage range.

Voh = High end of output voltage range.

V_{ol} = Low end of output voltage range.

In this case, Av= 3.

R1 and R2 are selected in the circuit to yield the desired gain. The input resistance R1 is assumed 100kthat is the input impedance. The resistors R1 and R2 determine the DC gain Av as *Av = R2/R1* and *R2= 3*100 = 300K*.

In order to scale the DC voltage input range Vbias is found as follows:

Vbias = ((Av V_{IH}) + V_{OL})/ (Av+1)*

Vbias = ((3 3.33) + -5)/ (3+1)*

Vbias = 1.25 v.

3.4.1.3 Calibrating the accelerometer output. The 12-bit ADC will output a decimal value in range of 0 to 4095 with 0 corresponding to 0v and 4095 corresponding to 3.3 v output of the accelerometer. Thus, the interpretation of any digital value say x obtained from ADC in terms of the accelerometer output voltage is given as follows:

*Accelerometer output voltage = (x*3.3)/4095*

At 3.3V, the 0g point is approximately 1.66V. The Accelerometer output voltage Xout is represented as ‘x’. Thus, with respect to 0g point the voltage is given as $y = x - 1.66$.The sensitivity at 3.3 v supply for the accelerometer is 333mV/g. Therefore, the acceleration in x direction is given as $(y/0.333)\ g$.

3.4.2 Trailer Cab Potentiometer. The test-bed consists of a potentiometer embedded into a socket in between the truck cab and trailer. The potentiometer is used to calibrate the trailer-cab angle whose measurements are critical to avoid jackknifing of the test-bed trailer truck. The potentiometer selected was a bourn 50KΩ precision potentiometer [22]. It provided accurate angle measurements.

The potentiometer had mechanical angle range from 0 degrees to 340 degrees. Thus, the trailer cab angles are calibrated in range from -170 degrees to + 170 degrees. Figure 3.10 shows the mechanical angle of the potentiometer arrangement.

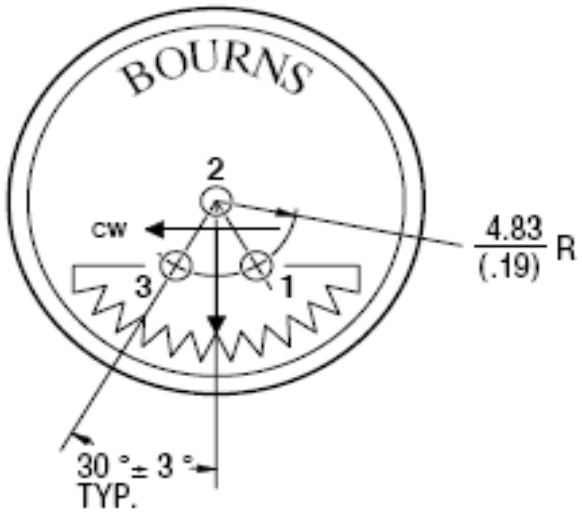

Figure 3.10: Potentiometer mechanical angles

The mechanical angle of potentiometer is 340 degrees while traversing from Point 3 to Point 1 in clockwise direction. Reference Point 2 is assumed 0 degrees where the trailer and cab are linear. Reference Point 3 is the point where the angle between trailer and cab is -170 degrees when cab movement is anticlockwise. Reference Point 1 is the

point where the angle between the trailer and the cab is +170 degrees when the cab movement is clockwise. The data acquisition software library always presents the trailer cab angle with respect to the 0-degree reference position.

With +5 V DC supply voltage at reference point 3 the potentiometer output is 0v DC and at reference point 1 the potentiometer output is 5 V DC and at reference point 2 the output is 2.5V DC. The 12-bit ADC will generate a decimal value in the range of 0 to 4095 with 0 corresponding to 0 V DC and 4095 corresponding to 5 V DC output of the potentiometer. Thus, the interpretation of any digital value say *x* obtained from ADC in terms of the potentiometer output voltage is given as Trailer cab angle in radians = (340*x/4095)-170. So at Reference Position 3 the output would be -170 degrees and at reference position 1 the output would be + 170 degrees.

Figure 3.11 shows the trailer cab potentiometer.

Figure 3.11: Trailer cab potentiometer

3.4.3 Test-bed Vehicle Speed Sensor. The test-bed features a Hewlett-Packard HEDS-5000 optical shaft encoder [23]. The shaft encoder outputs a square wave whose frequency is proportional to the speed of the motor shaft. The encoder output is directly connected to the 4I27 motor controller card.

The incremental shaft encoder operates by translating the rotation of the shaft into light interruptions in the form of electrical pulses. The light source is a light emitting diode channeled into a parallel beam of light by a molded lens. The light beam thus becomes an output pulse only when apertures in the phase plate wheel and shaft code wheel are lined up. Figure 3.12 shows the block diagram of the shaft encoder operation.

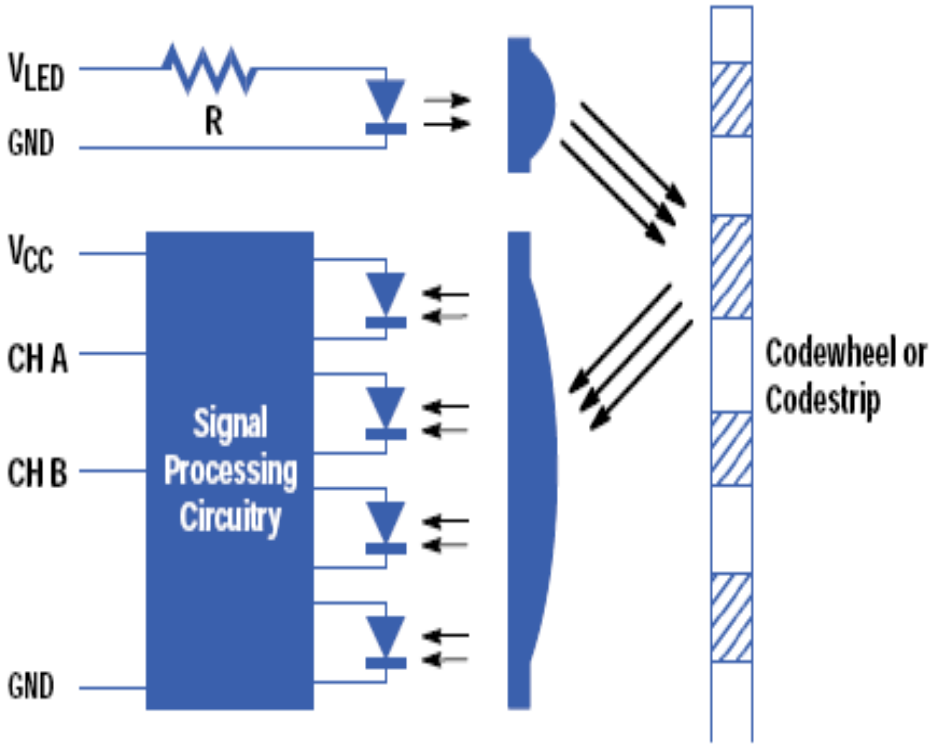

Figure 3.12: Block diagram of the optical shaft encoder operation

A code wheel rotates between the emitter and the detector and causes the light beam to be interrupted by the pattern of spaces and bars on itself. The photodiodes that detect these interruptions are arranged in a pattern that corresponds to the radius and the design of the code wheel.

These detectors are also spaced such that a light period on one pair of detectors corresponds to a dark period on the adjacent pair of detectors. The photodiode outputs are then fed through the signal processing circuitry. Comparators receive these signals and produce the final outputs for channels A and B. Due to this integrated phasing technique, the digital output of channel A is in quadrature with that of channel B (90 degrees out of phase). Figure 3.13 shows sample output waveforms of the encoder.

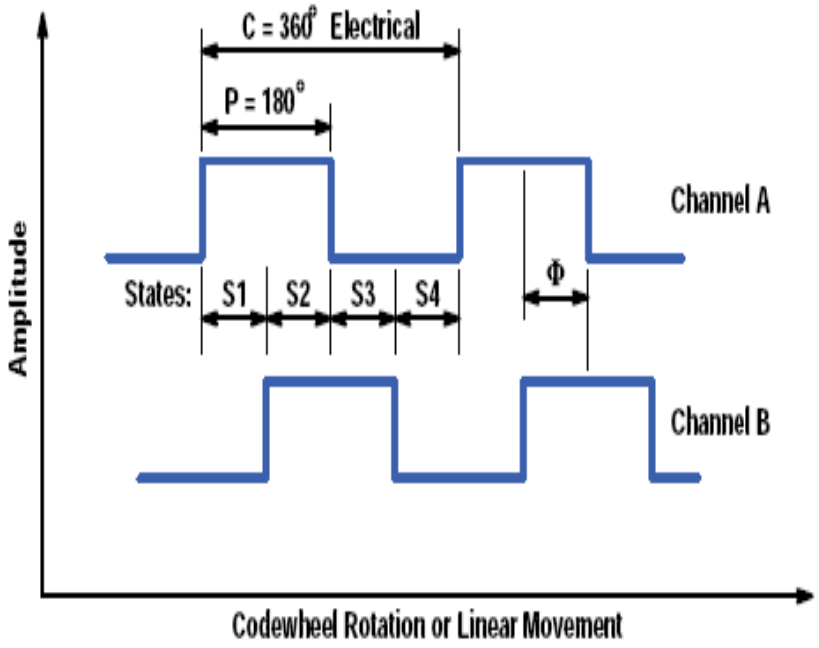

Figure 3.13: Output waveform of optical shaft encoder

3.4.4 Test-bed Vehicle Steering Angle Sensor. The steering angle is measured through the servo used to control the steering. The servo uses a potentiometer to perform the position control and the same potentiometer is used to find the steering angle. The steering angle output is used as feedback by the controller to make corrections to steering angle. Figure 3.14 shows the servo used to control steering angle.

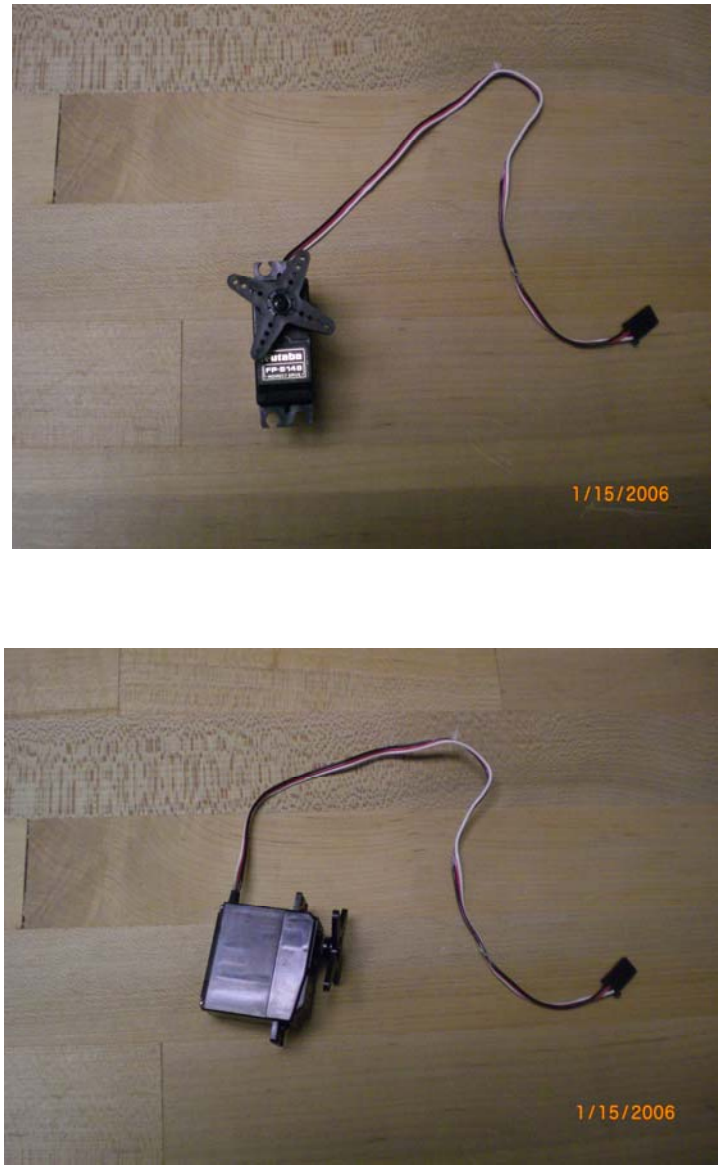

Figure 3.14: Steering servo

3.5. THE TEST-BED ACTUATORS

3.5.1 Driver Motor. The driver motor for moving the truck is a DC motor. The driver motor is connected to the 4I27 motor controller card that modulates the pulse width of the PWM pulses supplied to this motor to actuate its speed and direction. Figure 3.15 shows the arrangement of the driver motor from bottom view of the cab.

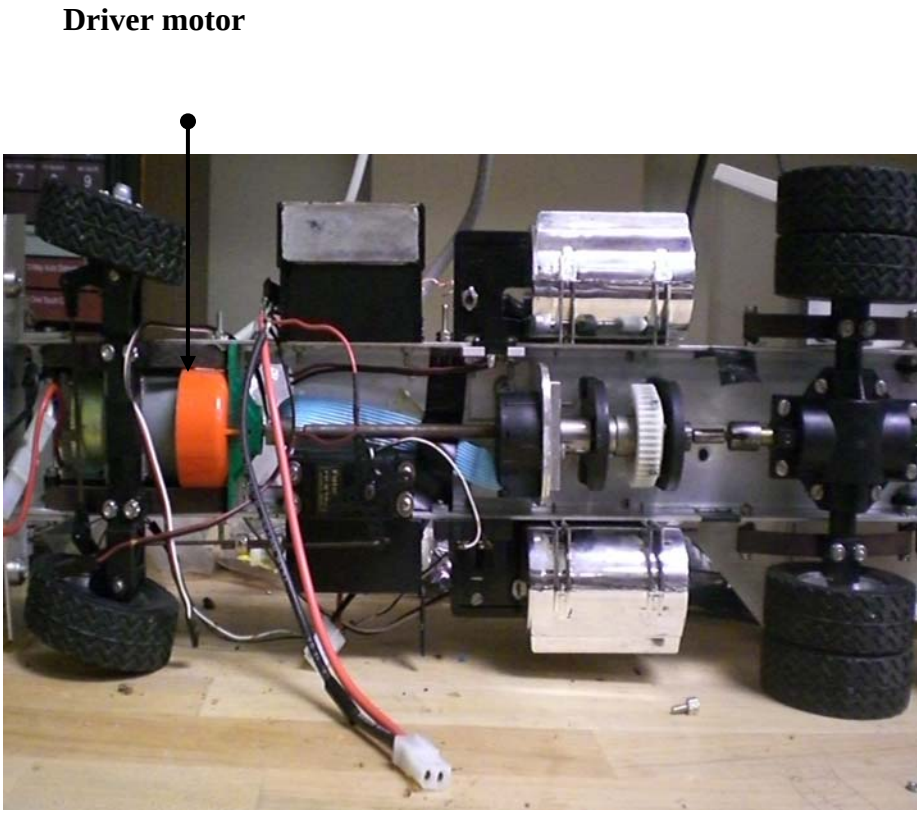

Figure 3.15: Steering and driver motor connection (bottom view)

3.5.2 Steering Motor. A DC motor is used to actuate the steering angle. The steering angle potentiometer updates the current steering angle, which is used to maneuver the nest steps in steering the truck. The steering motor is connected to 4I27

motor control card. Figure 3.16 shows the arrangement of steering motor (bottom view of test-bed vehicle cab).

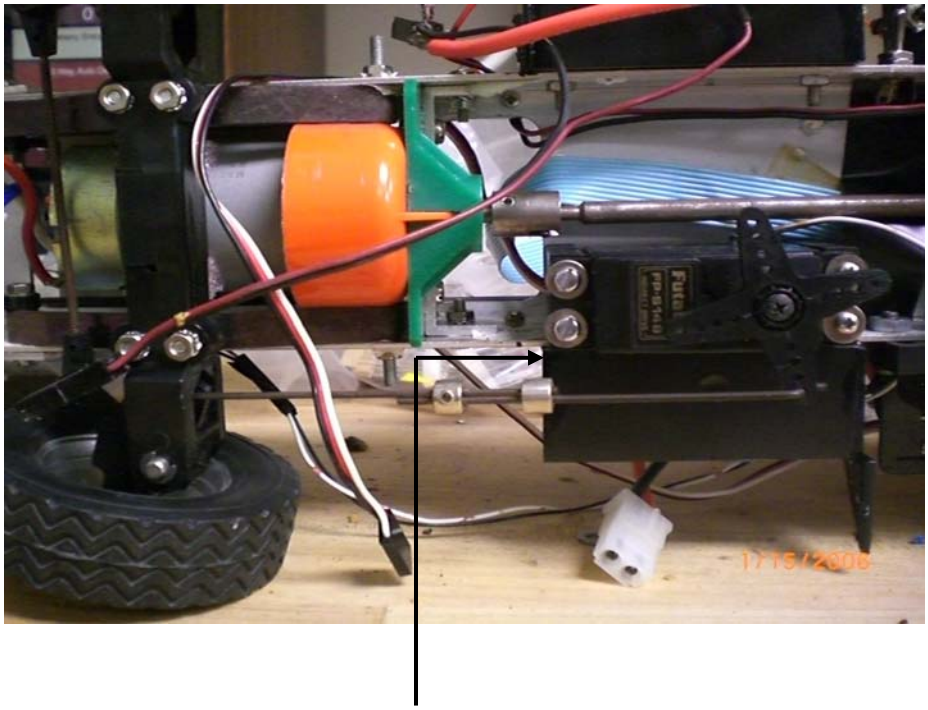

Figure 3.16: Steering motor

4. REAL TIME APPLICATION INTERFACE (RTAI)

RTAI stands for *Real Time Application Interface* for Linux as given in [11], [12], and [14]. RTAI lets the user develop applications with strict timing constraints. RTAI consists mainly of a patch to the Linux kernel that introduces a hardware abstraction layer and adds real time capabilities to the kernel generating a real time operating system within a non real time Linux environment. It provides ample programming services that can be utilized effectively in developing real time embedded system. It takes control of the hardware interrupts from the Linux kernel and adds real time precision to the interrupt handling.

Consequently, the Linux kernel act as a low priority task and is functional when CPU is not in demand by any real time task. The real time tasks are scheduled by RTAI and they can communicate with any of non-real-time or Linux kernel tasks. The hardware abstraction layer provided by RTAI minimizes the amount of changes to be made to kernel code and has features that make maintenance of RTAI quite easy. The test-bed single board computer was embedded with RTAI Linux based operating system thereby adding real time properties to the system.

The interrupts for a periodic application can be designed using two timer tick periods supported by RTAI i.e. “periodic” and “one-shot”. In the periodic mode, the RTAI sets its timer to interrupt at the desired period. Interrupt occurs when timer overflows and RTAI timer is reloaded again with the same time requirement. This method ensures that there is no CPU overhead spent in servicing the timer.

On the other hand, in one shot mode RTAI reprograms the timer at every interrupt. Hence, considerable CPU overhead is spent using the one-shot mode. Figure 4.1 shows RTAI architecture and shows how the RTAI forms an intermediate layer in between the hardware and the Linux kernel. Figure 4.1 also briefs out the way real time tasks defined through the RTAI application programming interface and the non real time ordinary Linux kernel tasks communicate amongst themselves. The real time tasks do not communicate directly with the Linux kernel and normally do not make any ordinary Linux kernel function calls.

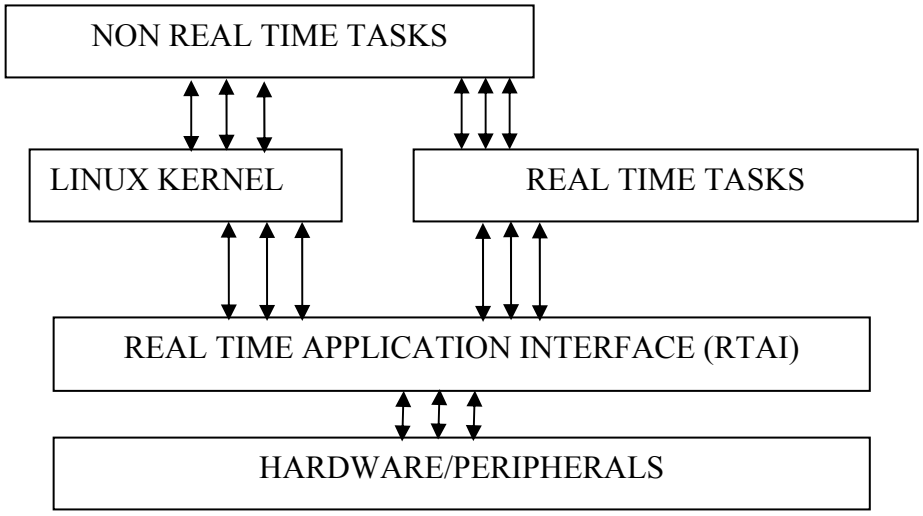

Figure 4.1: RTAI architecture

RTAI is based on the concept of *Real-Time Hardware Abstraction Layer (RTHAL)* that is used for intercepting the hardware interrupts and processing them. RTHAL minimizes the number of changes required to be made to kernel code and assigns pointers to kernel data and functions operating on the system hardware thereby forming a so-called layer in between hardware and kernel. RTHAL allows easy access and changes to interrupt handlers without the need to make complex arrangements to kernel structure.

The RTHAL maintains an interrupt handler table containing functions that are called for handling different interrupts. When RTAI is patched the hardware interaction of the kernel is replaced by the pointers pointed by RTHAL and hardware interaction and interrupt handling is done completely by RTAI. RTAI provides hard real time capabilities in both kernel space as well as user space. RTAI provides hard real time capabilities through two schedulersthat now are called rtai_lxrt and rtai_sched. The user space scheduling is available for Linux processes and threads. The rtai_sched allows real time scheduling of RTAI kernel tasks along with scheduling of objects that can be scheduled by Linux such as Linux processes and Linux threads.

4.1. INSTALLING RTAI ON TEST-BED CPU

The Arcom control systems manufactured test-bed CPU came pre loaded with debian-based Linux 2.6 version. The process of patching RTAI to the existing Linux kernel was distributed in three tasks as explained.

4.1.1 Setting up Kernel Sources. The Kernel source for the target board can be found on the AEL Host environment on the host machine. On the test-bed single board computer, the path to the source was "*/opt/arcom/src/linux-source-2.6.16.14-arcom1.tar.gz"*. The kernel source and RTAI source were both extracted to "*/usr/src"*. A symlink "*/usr/src/linux*" pointing to the kernel source was created using the following command "*# ln -s /usr/src/linux-source-2.6.16.14-arcom1 /usr/src/linux"*. The RTAI was then patched to the kernel using the "*# patch -p1 /usr/src/rtai-3.4-cv/base/arch/i386/patches/hal-linux-2.6.16-i386-1.3-08.patch"* .The sources are now setup and next step was configuring and compiling the kernel and RTAI.

4.1.2 Configuring and Compiling the Kernel. The default configuration was obtained by executing "*# make ARCH=i386 sbc-gx533_defconfig"*. The options for real time kernel configuration were selected using following command "*# make ARCH=i386 xconfig"*. According to the RTAI installation guide, you need to turn off module versioning, ACPI support and APM support as these conflicts with RTAI. These features were turned off and the kernel was made pre-emptible. The kernel is then compiled using *"# ael-kernel-build --ARCH=i386 image"*. This compiled the kernel, built the image and built three .deb packages at "*/usr/src/"* which were installed on the test-bed CPU using the command "*# dpkg -i linux-image-2.6.16.14-arcom1_10.0.Custom_i386.deb"* .The test-bed single board computer was accessed using serial connection and using red boot command window the default kernel was changed to newly installed kernel as follows :

"*Red Boot> alias kernel /boot/vmlinuz- 2.6.16.14-arcom1*"

This made the test-bed single board computer ready to be installed with RTAI.

4.1.3 Configuring and Compiling RTAI. The working directory was changed to "*/usr/src"* and a new directory was created called build. The working directory was

changed to "/*usr/src/build*" and the following command was used to build RTAI:

"*# make -f /usr/src/rtai-3.4-cv/makefile xconfig CC=i386-linux-gcc CXX=i386-linux-g++*". The configuration features were saved and RTAI was compiled by executing "*# make all*". The RTAI was installed to a temporary folder using "*# make install DESTDIR=/tmp/rtai*" .The installed source was compressed using "*# cd /tmp/rtai*" and then "*# tar -cvzf... /rtai_install.tar.gz*". RTAI was installed in the test-bed CPU using the following command "*# tar -xvzf /root/rtai_install.tar.gz*". This concluded the installation of RTAI.

5. TEST-BED SOFTWARE LIBRARY

The test-bed software contains the real time software library that consists of real time threads for sensing and real-time control functions for actuation. The threads periodically update the sensor readings at one millisecond rate to individual global arrays corresponding to each sensor. The global arrays always keep a track of the previous five sensor values along with the current value, thus providing the access to any of the five previous values for each sensor.

The arrays keep track of the previous five sensor values before being overwritten by recent values. All the sensor update threads are equal priority threads and would require the library user to call an initialization routine to start these threads concurrently.

First, the threads update the current sensor value and then they sleep for one millisecond until the next interval of update. The SBC central processing unit (CPU) is available during this interval for other computations. The real time software ensures that the sensor values keep being updated in the background periodically independent of the complexity of the application software and CPU overhead caused.

The control functions are real time and ensure that the control action is processed without any significant latency. The control functions include position, steering, and speed control functions as well as emergency functions.

The test-bed software library provides an Application Programming Interface (API) to the user where the user can easily use the library functions and global variables updated by the library to get the current state of the vehicle, make decisions and implement the desired control action. RTAI runs in kernel space with the highest priority. The real time task that it manages also runs in kernel space.

However, kernel space application development faces problems in using source level debugger and fatal programming errors in kernel space such as allocating restricted memory can crash the whole kernel system. However, RTAI provides a solution to this problem. RTAI provides LXRT extension ([14], [15]), where the application can be developed as normal user application. All the LXRT tasks are created as normal user space tasks. All the tasks are initialized as real time tasks using an initialization routine

call provided by LXRT module. The initialization creates a friend task that runs on behalf of the user space task in kernel space. Whenever LXRT task calls are made, the friend task executing in kernel space executes the real function.

The software library is developed in user space using LXRT module. All the functions in the library are initialized as LXRT tasks and they are executed in user space. The user of this library would just be required to include the library during compilation and the real time application developed by the user would be executed in the user space. Figure 5.1 explains the test-bed software library and the user application implementation in user space.

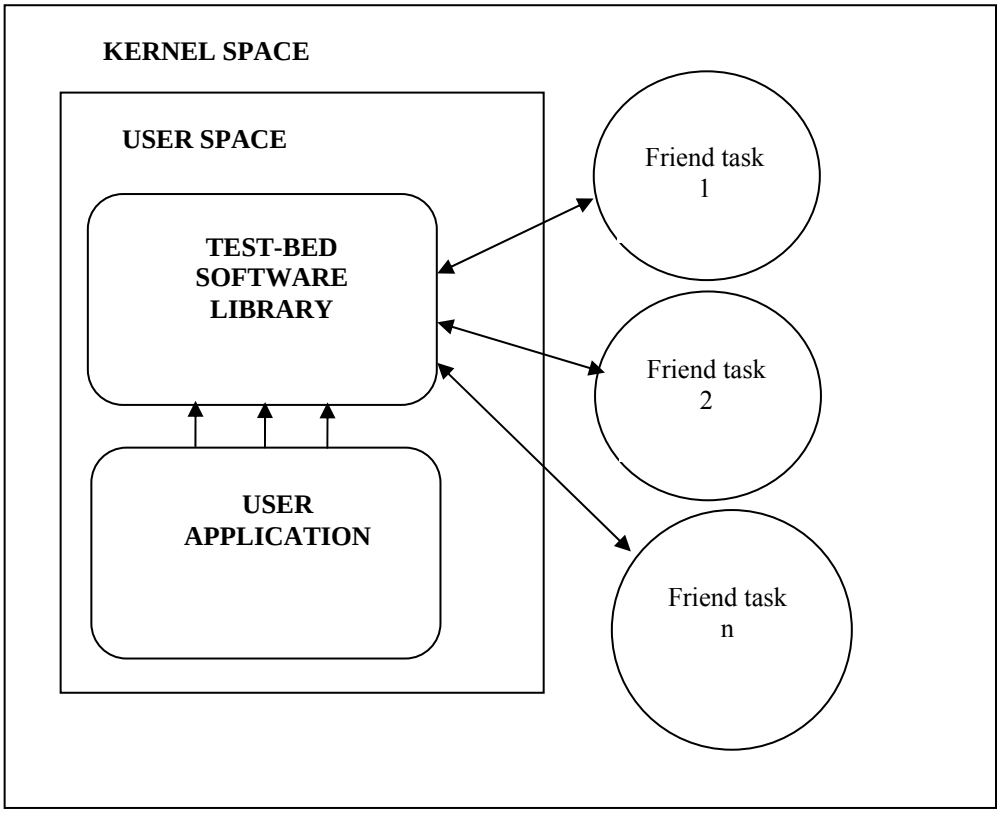

Figure 5.1: Test-bed software library and user application implementation

The test-bed software library components are described in detail in following sections:

5.1. GLOBAL VARIABLES HOLDING THE SENSOR VALUES

The sensor values are updated every one millisecond to global arrays that are allocated dynamically when the user calls the initialization routine for the library from his application. Each sensor updates to a separate global array and at any point of time the arrays hold the five previous sensor updates. Pointers are initialized to each global array and the sensor values of each instant stored can be accessed through index addressing.

The dynamic memory allocation function call *calloc ()* is used to assign memory dynamically for six update instants for each sensor. The pointers point to the initial location of allocated chunk of memory. The allocation can be thought of as allocating dynamic memory array for storing the current update of each sensor along with five previous updates for each sensor.

The dynamic memory implementation also ensures that memory is allocated at run time only after the initialization routine is invoked by the user. The user has to ensure that the initialization routine is not called more than once. The pointers are listed out as follows:

Function: Pointer holding acceleration value in the x direction.
Synopsis: truck_accel_x
Description: Pointer pointing to updates of the trucks acceleration along x direction of the accelerometer. The updates are the truck acceleration along the x direction of the accelerometer. Indexed addressing of the pointer can access sensor values.

Function: Pointer holding acceleration value in the y direction.
Synopsis: truck_accel_y
Description: Pointer pointing to updates of the trucks acceleration along y direction of the accelerometer. The updates are the truck acceleration along the y direction of the

accelerometer. Indexed addressing of the pointer can access sensor values instantly in real time.

Function: Pointer holding acceleration value in the z direction
Synopsis: truck_accel_z
Description: Pointer pointing to updates of the trucks acceleration along z direction of the accelerometer. The updates are the truck acceleration along the z direction of the accelerometer. Indexed addressing of the pointer can access sensor values.

Function: Pointer holding the truck potentiometer value
Synopsis: truck_pot
Description: Pointer pointing to updates of the truck potentiometer sensor for the angle between trailer and cab in radians. Indexed addressing of the pointer can access sensor values.

Function: Pointer holding the truck steering angle value
Synopsis: truck_steer
Description: Pointer pointing to updates of the trucks steering angle in radians. Indexed addressing of the pointer can access sensor values.

Function: Pointer holding value of the speed of truck
Synopsis: truck_pot
Description: Pointer pointing to updates of the truck speed in RPM. Indexed addressing of the pointer can access sensor values.

The library users do not have direct access to these pointers and the users should design their application using the test-bed library in such a way that the application should ensure that these pointers are not accessed by the application. Any wild usage of these pointers could result in critical damages.

Damages may include the pointers corrupting the kernel data and data structures on which the kernel operatesthat can cause fatal crashing of the kernel. The user

application has to ensure that there is no wild usage of pointers in the application that can corrupt the memory accessed by the test-bed library. It would be advisable to follow safe programming practices.

5.2. REAL TIME THREADS UPDATING THE SENSOR VALUES

The test-bed library API provides real time threads (user space POSIX pthreads) which periodically update the sensor values to global arrays corresponding to each individual sensor. All the threads are defined as real time threads and the user is required to include an initialization routine to start the real time threads. The task priority, stack size, data to be passed and the other essential task related data are passed through the initialization routine for the real time task.

The threads run concurrently and the real time design ensures that any sensor update event by any of these threads is never missed irrespective of the load on CPU. The RTAI LXRT module allows declaring normal threads in user space as real time threads using same application programming interface. Using RTAI API all these threads are defined as periodic updating threads with one millisecond update interval and are they assigned equal priorities.

The threads have equal priority and are scheduled in round robin format by the real time application interface scheduler. The threads share the ADC driver function in accessing the channel updates for each sensor. To avoid common resource sharing problems a global mutex is locked by each thread to before accessing the ADC function and unlocked after accessing the ADC function.

The sensor update threads are listed as follows:

Function: Thread updating truck acceleration value along x direction

Synopsis: truck_accelerometer_x

Description: Real time threadthat updates the truck accelerometer sensor readings in x direction to the global array pointed by truck_accel_x. The updates are the truck acceleration along the x direction of the accelerometer.

Function: Thread updating truck acceleration value along y direction

Synopsis: truck_accelerometer_y

Description: Real time thread that updates the truck accelerometer sensor readings in y direction to the global array pointed by truck_accel_y. The updates are the truck acceleration along the y direction of the accelerometer.

Function: Thread updating truck acceleration value along z direction

Synopsis: truck_accelerometer_z

Description: Real time thread that updates the truck accelerometer sensor readings in z direction to the global array pointed by truck_accel_z. The updates are the truck acceleration along the z direction of the accelerometer.

Function: Thread updating truck potentiometer value

Synopsis: truck_potentiometer

Description: Real time thread that updates the truck potentiometer sensor readings for angle between trailer and cab in radians to the global array pointed by truck_pot.

Function: Thread updating truck steering angle value

Synopsis: truck_steering_motor

Description: Real time thread that updates the truck steering angle sensor readings in radians to the global array pointed by truck_steer.

Function: Thread updating current state value

Synopsis: Current_state_k

Description: Real time thread that updates the current state of sensor update to global variable current_state. The states are updated up to $_{10}$**9** instants before reinitializing itself back to zero.

The threads update the sensor values and do not communicate with any other function in the library. The functions to obtain the sensor updates done by these threads are explained in Section 5.3.

5.3. REAL TIME SENSOR UPDATE ACCESS FUNCTIONS

The test-bed library API provides various functions to access sensor values at any instant of each sensor. Given the previous time step, these functions return the corresponding previous value of the respective sensor. For example to access the 3rd previous value of the truck sensor the function call would be in form of "truck_sensor_update (3)" and current update of the sensor can be accessed by "truck_sensor_update (0)".

This methodology avoids the direct user interaction of the global arrays and thus prevents the user from corrupting the library data. Without these functions, the user may be required to access the values pointed by the global array pointers for the sensor values and significant programming effort and caution to correctly access the sensor update values would be required. Any programming error in the use of these pointers would result in corrupted data and disastrous control decision. This could also lead to system instability due to inefficient control implementation.

The update functions are defined to work in real time, and they run in user space using LXRT module based upon same principle for the sensor update threads. This methodology also ensures user-friendly implementation since the sensor value access is reduced to mere function calls.

The sensor update functions mainly take in the previous required instance of update as argument and output the sensor value at that instant. The sensor update access functions provided by the test-bed library API are explained in detail as follows:

Function: Function to access potentiometer update
Synopsis: float truck_potentiometer_update (int n)
Description: Real time function that updates the nth previous truck potentiometer update

to get the trailer cab angle in radians. This reading is useful to avoid landing of the test-bed truck in jack-knife position where it cannot move.

Function: Function to access steering angle update
Synopsis: float truck_steering_update (int n)
Description: Real time function that updates the nth previous truck steering angle update in radians.

Function: Function to access truck speed update
Synopsis: float truck_speed_update (int n)
Description: Real time function that updates the nth previous truck speed sensor update to get the truck speed in RPM.

Function: Function to access truck acceleration along x direction
Synopsis: float truck_accelerometer_x_update (int n)
Description: Real time function that updates the nth previous truck accelerometer update along x direction of the accelerometer. The updates are the truck acceleration along the x direction of the accelerometer.

Function: Function to access truck acceleration along y direction
Synopsis: float truck_accelerometer_y_update (int n)
Description: Real time function that updates the nth previous truck accelerometer update along y direction of the accelerometer. The updates are the truck acceleration along the y direction of the accelerometer.

Function: Function to access truck acceleration along z direction
Synopsis: float truck_accelerometer_z_update (int n)
Description: Real time function that updates the nth previous truck accelerometer update along z direction of the accelerometer. The updates are the truck acceleration along the z direction of the accelerometer.

Function: Function to access truck velocity in encoder counts

Synopsis: long ReadRealVelocity (Motor *motorx)

Description: Real time function that updates the current truck speed. Using this function would require assigning the driver motor to motor0 using the motor assign function explained in Section 5.4.1.

5.4. REAL TIME CONTROL FUNCTIONS

The test-bed library API provides a vast number of real time control functions to actuate the steering angle control and speed control motors. The user can access the sensor updates and accordingly use any of these control function calls and implement the control decision as per the control algorithm they are using. The motor controller driver functions are defined to be real time functions and thus RTAI takes charge of the Motor controller peripheral and communicates directly with the actuators.

The control functions include functions for controlling the motor speed and position and to load the target velocity, target position and PID filter parameters. The control functions also include emergency procedures for stopping the motor smoothly and abruptly.

The real time sensor value threads provide real time status of current and previous five sensor values. These sensor values are indicative of the current state of the vehicle and depending upon the current state the control algorithm determines the desired next state of the vehicle. The control functions are then used to achieve the desired next state of the vehicle by actuating the vehicle actuators.

The control functions communicate with the LM629 processors on the 4I27 motor controller card. The control functions access data structures declared in the software library. The data structures used and the control functions are described in detail.

Function: Function to initialize motor

Synopsis: void AssignMotor (Motor *motorx, int card, int whichhalf)

Description: This function is used to assign motor to one of the half's (one of the two

LM629 processors onboard) of the card. The argument passed is a pointer to the data structure Motor declared in the software library and the card address (ox200 for the test-bed card) and the half of the card. This function is to be implemented as part of initialization procedure.

Example: To assign the driver motor to define driver motor as motor 0 and assign it to first half of 4I27 card the following procedure is implemented:

```
Main ()
{
        Motor motor0; //part of initialization
        AssignMotor (&motor0,0x200,0);
        …..
}
```

Function: Function to turn on motor

Synopsis: void TurnOnMotor (Motor *motorx)

Description: This function is used to turn on the motor under control of 4I27 motor controller card. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the motor start command declared in the library header file, to the 4I27 motor controller card.

Example: To turn on driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{
        …..                    //initialization
        TurnOnMotor (&motor0);
        ….. }
```

Function: Function to turn off motor

Synopsis: void TurnOffMotor (Motor *motorx)

Description: This function is used to turn off the motor under control of 4I27 motor

controller card. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the motor stop command declared in the library header file, to the 4I27 motor controller card.

Example: To turn off driver motor defined as motor zero the following procedure is implemented:

```
Main ()
{
        …..                    //initialization
        TurnOffMotor (&motor0);
        …..
}
```

Function: Function to smoothly stop the motor

Synopsis: void StopSmoothly (Motor *motorx)

Description: This function is used to smoothly stop the motor under control of 4I27 motor controller card. The argument passed is a pointer to the data structure Motor declared in the software library. This function can be used to stop the vehicle smoothly on completion of any of desired trajectory of position, velocity or acceleration. This function passes the command to smoothly stop the motor declared in the library header file, to the 4I27 motor controller card.

Example: To smoothly stop the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{       …..                    //initialization
        StopSmoothly (&motor0);        …..
}
```

Function: Function to abruptly stop the motor

Synopsis: void StopAbruptly (Motor *motorx)

Description: This function is used to abruptly stop the motor under control of 4I27 motor controller card. The argument passed is a pointer to the data structure Motor declared in the software library. This function can be used to stop the vehicle smoothly on completion of any of desired trajectory of position, velocity or acceleration. This function passes the command to abruptly stop the motor declared in the library header file, to the 4I27 motor controller card.

Example: To abruptly stop the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{
        …..                    //initialization
        StopAbruptly (&motor0);
        …..
}
```

Function: Function to define home position

Synopsis: void DefineHome (Motor *motorx)

Description: This function defines the current position as home position. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the command to define current position as home positionthat is declared in the library header file, to the 4I27 motor controller card.

Example: To define the current position of the driver motor defined as motor 0 as home position the following procedure is implemented:

```
Main ()
{       …..                    //initialization
        DefineHome (&motor0);
        …..     }
```

Function: Function to set index position

Synopsis: void SetIndexPosition (Motor *motorx)

Description: This function is used to define current position as Index position. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the command to define current position as Index positionthat is declared in the library header file, to the 4I27 motor controller card.

Example: To define index position for the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{       …..                     //initialization
        SetIndexPosition (&motor0);
        …..
}
```

Function: Function to set position error to cause software interrupt

Synopsis: void LoadPosErrorForInt (Motor *motorx, int maxerr)

Description: This function is used to load the position errorthat can trigger off a software interrupt. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of maximum error that should trigger off the interrupt. This function passes the command to load the position errorthat is declared in the library header file and the position error, to the 4I27 motor controller card.

Example: To define 1000 encoder counts as the maximum error to cause interrupt for the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{
        …..                     //initialization
        LoadPosErrForInt (&motor0, 1000);
        …..
}
```

Function: Function to set position error to stop the motor

Synopsis: void LoadPosErrorForStop (Motor *motorx, int maxerr)

Description: This function is used to load the position errorthat can trigger off stopping of the vehicle. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of maximum error that should trigger off the stopping of vehicle. This function passes the command to load the position errorthat is declared in the library header file and the position error, to the 4I27 motor controller card.

Example: To define 1000 counts as the maximum error to stop the motor for the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{       …..                     //initialization
        LoadPosErrForStop (&motor0, 1000);
                …….
                }
```

Function: Function to set absolute breakpoint

Synopsis: void SetBreakpointAbsolute (Motor *motorx, long position)

Description: This function is used to set the absolute position breakpoint for motion. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of position that should be set as breakpoint. This function passes the command to load the absolute position break pointthat is declared in the library header file and the absolute motion breakpoint, to the 4I27 motor controller card.

Example: To define 1000 encoder counts as the absolute breakpoint for the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{       …..                     //initialization
        SetBreakpointAbsolute (&motor0, 1000);
        …..
}
```

Function: Function to set relative breakpoint

Synopsis: void SetBreakpointRelative (Motor *motorx, long position)

Description: This function is used to set the relative position breakpoint for motion. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of relative position that should be set as breakpoint. This function passes the command to load the relative position break pointthat is declared in the library header file and the relative position breakpoint, to the 4I27 motor controller card.

Example: To define 1000 encoder counts as the relative breakpoint for the driver motor defined as motor 0 the following procedure is implemented:

```
Main ()
{
        …..                         //initialization
        SetBreakpointRelative (&motor0, 1000);
        …..     }
```

Function: Function to load PID filter parameters.

Synopsis: void LoadFilterParameters (Motor *motorx).

Description: This function is used to load the filter coefficients from their pre-loaded buffers. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the command to load the filter parametersthat is declared in the library header file, to the 4I27 motor controller card. The filter parameters can be pre loaded using any of filter parameter load functions and are loaded using this function before run time.

Example: To define filter parameters KP,KI,KD,IL for the 4I27 PID filter as 200,100,2000,10 respectively and assign them to data structure pointer motorx,the data structure pointer is passed to an initialization routine which can initialize the filter parameters and use “LoadFilterParameters” function to load them as explained in the example.

```
initfunction (Motor *motorx)
{       Motorx->Filter.KP = 200;       // proportional gain
```

```
        Motorx->Filter.KI = 100;        // integral gain
        Motorx->Filter.KD = 2000;       // derivative gain
        Motorx->Filter.IL = 10;         // integration limit
        LoadFilterParameters(motorx);
}
```

The initfunction routine can be used as an initialization routine and is one of the important steps in setting up the PID filter coefficients for the 4I27 motor card. The function should be following the reset function for the 4I27 card.

The reset function should always precede the initfunction routine during run time whenever the PID filter parameters of the 4I27 are desired to be changed. The reset function resets the 4I27 and the onboard LM629 processors and allows any changes to the 4I27 and LM629 processors possible.

Function: Function to load proportional gain

Synopsis: void LoadKP (Motor *motorx, unsigned int KP)

Description: This function is used to set the proportional gain KP for the PID filter of 4I27. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of KP. This function passes the command to load the proportional gainthat is declared in the library header file and the proportional gain, to the 4I27 motor controller card PID filter. This function can be used as one of the testing function while tuning the PID filter loop of the LM629 processor on the 4I27 motor controller card.

Example: To define proportional gain as 100 for PID filter with respect to driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                         //initialization
        LoadKP (&motor0, 100);
        …..
}
```

Function: Function to load Integral gain

Synopsis: void LoadKI (Motor *motorx, unsigned int Ki)

Description: This function is used to set the integral gain Ki for the PID filter of 4I27. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of Ki. This function passes the command to load the integral gainthat is declared in the library header file and the integral gain, to the 4I27 motor controller card PID filter.

Example: To define integral gain as 50 for PID filter with respect to driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        LoadKI (&motor0, 50);
        …..
}
```

Function: Function to load Derivative gain

Synopsis: void LoadKD (Motor *motorx, unsigned int KD)

Description: This function is used to set the derivative gain KD for the PID filter of 4I27. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of KD. This function passes the command to load the derivative gainthat is declared in the library header file and the derivative gain, to the 4I27 motor controller card PID filter.

Example: To define derivative gain as 2000 for PID filter with respect to driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        LoadKD (&motor0, 2000);
        …..  }
```

Function: Function to load Integral limit

Synopsis: void LoadIL (Motor *motorx, unsigned int IL)

Description: This function is used to set the integral limit IL for the PID filter of 4I27. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of IL. This function passes the command to load the integral limitthat is declared in the library header file and the integral limit, to the 4I27 motor controller card PID filter.

Example: To define integral limit as 10 for PID filter with respect to driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                     //initialization
        LoadIL (&motor0, 50);
        …..
}
```

Function: Function to update PID filter coefficients

Synopsis: void UpdateFilter (Motor *motorx)

Description: This function is used to update the filter coefficients from their pre-loaded buffers. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the command to update the filtersthat is declared in the library header file, to the 4I27 motor controller card. The filter parameters can be pre loaded using any of filter parameter load functions and can be updated using this function at run time.

Example : To define filter parameters KP,KI,KD,IL for the 4I27 PID filter as 200,100,2000,10 respectively and assign them to data structure pointer motorx,the data structure pointer motorx is passed to an initialization routine which can initialize the filter parameters and use “LoadFilterParameters” function to load them. The filter parameters are then updated using “UpdateFilter” function as explained in the example.

This function can be used at runtime to update PID filter parameters depending upon the control demands. At run time, any undesired change in motion can cause to change the desired control action and this function can be used then to change the PID filter parameters at run time.

```
initfunction (Motor *motorx)
{
        Motorx->Filter.KP = 200;       // proportional gain
        Motorx->Filter.KI = 100;       // integral gain
        Motorx->Filter.KD = 2000;      // derivative gain
        Motorx->Filter.IL = 10;        // integration limit
        LoadFilterParameters(motorx);
        UpdateFilter (motorx);
}
```

Function: Function to load trajectories

Synopsis: void LoadTrajectory (Motor *motorx, Trajectory *traj)

Description: This function is used to load the trajectories of Acceleration, Velocity and Position to the 4I27 motor controller card. The argument passed is a pointer to the data structure Motor declared in the software library and a pointer to the data structure Trajectory declared in the software library. This function passes the command to load the trajectorythat is declared in the library header file, and the trajectories (Acceleration, Velocity, and Position) which are defined through trajectory load commands.

This function can be used at runtime to update motion trajectories depending upon the control demands. At run time, any undesired change in motion can cause to change the desired control action and this function can be used then to change the trajectories at run time.

Example: To define trajectories of position, velocity and acceleration as zero encoder counts, 1000 encoder counts/sampling period and 100 counts/ (sampling period*sampling period) respectively the procedure is explained as follows:

```
Main ()
{
        …                                   //initialization
        Motor motor0;
        Trajectory trajx;
        trajx.Position = 0;
        trajx.Velocity = 1000;
        trajx.Acceleration = 100;
        LoadTrajectory (&motor0, &trajx);
        ……
}
```

Function: Function to load acceleration trajectory

Synopsis: void LoadAcceleration (Motor *motorx, unsigned long acceleration)

Description: This function is used to load the acceleration set point for motion. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of acceleration that should be set as set point. This function passes the command to load the accelerationthat is declared in the library header file and the acceleration, to the 4I27 motor controller card. The acceleration is always loaded in terms of the encoder counts since the 4I27 functions deal with the trajectories in terms of encoder counts. The user of the test-bed library has to calibrate the trajectories accordingly to the desired trajectory range.

Example: To load acceleration as 100 encoder counts/ (Sampling period) for driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                         //initialization
        LoadAcceleration (&motor0, 100);
        …..
}
```

Function: Function to load velocity trajectory

Synopsis: void LoadVelocity (Motor *motorx, unsigned long velocity)

Description: This function is used to load the velocity set point for motion. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of velocity that should be set as set point. This function passes the command to load the velocitythat is declared in the library header file and the velocity, to the 4I27 motor controller card.

Example : To load velocity as 1000 encoder counts/sampling period for driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                     //initialization
        LoadVelocity (&motor0, 1000);
        …..
}
```

Function: Function to load position trajectory

Synopsis: void LoadPosition (Motor *motorx, unsigned long position)

Description: This function is used to load the position set point for motion. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of position that should be set as set point. This function passes the command to load the positionthat is declared in the library header file and the position, to the 4I27 motor controller card.

Example: To define position as 0 encoder counts (velocity mode) for driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                     //initialization
        LoadPosition (&motor0, 0);
        …..     }
```

Function: Function to start pre loaded trajectory

Synopsis: void StartTrajectory (Motor *motorx)

Description: This function is used to start a pre-loaded trajectory using any of trajectory load functions. The argument passed is a pointer to the data structure Motor declared in the software library. This function passes the command to start a pre-loaded trajectorythat is declared in the library header file, to the 4I27 motor controller card.

Example: To define position as zero encoder counts (velocity mode) for driver motor defined as motor 0 and start the trajectory the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        LoadPosition (&motor0, 0);
        StartTrajectory (&motor0);
        …..
}
```

Function: Function to set sampling interval

Synopsis: void SetSamplingInterval (Motor *motorx, unsigned int sinterval)

Description: This function sets the sampling interval of the 4I27 processor clock. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of desired sampling interval .This function is used as part of initialization procedure for the 4I27 motor controller card. The sampling interval supplied can be from zero to 255 and sampling interval is determined as SP = SAMPLING PERIOD = 256 uSec * (1+SI) with 8 MHz clock.

Example: To define minimum sampling interval for driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        SetSamplingInterval (&motor0, 0); //
        …..     }
```

Function: Function to set motion mode

Synopsis: void SetMotionMode (Motor *motorx, int mmode)

Description: This function sets the motion mode of the motor. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of motion modethat is already defined in the library header. This function is used as part of initialization procedure for the 4I27 motor controller card.

Example: To set position mode for driver motor defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        SetMotionMode (&motor0, PositionMode); // set up motor 0 for
                                                    Position mode
        …..
}
```

Function: Function to set encoder lines

Synopsis: void SetEncoderlines (Motor *motorx, int lines)

Description: This function sets the encoder lines for the respective motor. The argument passed is a pointer to the data structure Motor declared in the software library and the integer value of lines of encoderthat are 256 for the test-bed motor speed encoder. This function is used as part of initialization procedure for the 4I27 motor controller card.

Example : To set encoder lines equal to 500 for driver motor (which happens to be 500 in case of test-bed driver motor) defined as motor 0 the following procedure is implemented.

```
Main ()
{
        …..                    //initialization
        SetEncoderLines (&motor0, 500); //
        …..
}
```

Function: Function to select clock frequency

Synopsis: void SetClockFrequency (Motor *motorx, long frequency)

Description: This function is used to set the clock frequency of LM629. The argument passed is a pointer to the data structure Motor declared in the software library and the frequency required to be set.

Example: To set 8 MHz clock frequency for driver motor defined as motor 0 the procedure would be as follows:

```
Main ()
{
        …..                          //initialization
        SetClockFrequency (Motor0, 8000000);        //8 MHz clock
        …..
}
```

To set 4Mhz frequency for driver motor incase if the driver motor is changed to one which requires lower chopping frequency the function call in such a case would be

```
SetClockFrequency (Motor0, 4000000);        //4 MHz clock
```

5.5. TEST-BED SOFTWARE LIBRARY USAGE

The users of test-bed would be required to compile their application and link it to the test-bed software library. The user would also be required to call a routine called as "init_threads" as a part of the initialization routine.

Synopsis: init_threads (int *fdc)

Description: The init_thread routine initializes all the sensor update threads and starts them as periodic threads of one millisecond frequency. The argument supplied to init_thread routine is a pointer to the integer file handle for the ADC access function. The file handle is obtained by opening the ADC device node as follows: /dev/arcom/aim104/multi-io/0.

Example: The device node can be opened by using the open function as explained in the example. The open function returns the integer device file handlethat is supplied, to the

real time threadsthat use this device file handle to access the ADC device. This initialization routine is the first part of initialization that the user needs to call as a part of initialization procedure.

```
Main ()
{
        int *ptr;
        int a = open (/dev/arcom/aim104/multi-io/0, O_RDWR);
        ptr = &a;
        init_thread(&ptr);          //initialize all threads
        …..
}
```

The test-bed library header file is called as “testbedheader.h” and the user would be required to include the header file in his application. The header file includes following header files

- #include <stdio.h>
- #include <libaim104.h>
- #include <math.h>
- #include <time.h>
- #include <linux/module.h>
- #include <rtai.h>
- #include <rtai_sched.h>
- #include <rtai_fifos.h>
- #include <rtai_lxrt.h>
- #include <pthread.h>
- #include <stdlib.h>

Inclusion of the test-bed header file will automatically include the source code for all the header files mentioned. If the user intends to use any other library, the user would be required to include the respective header file in his application.

6. TEST-BED DESIGN PERFORMANCE

The RTAI kernel provides standard test suites to test the system performance for latency, preemption and context switching under heavy CPU load conditions [17]. The RTAI performance on test-bed CPU was tested for latency, preemption and context switching by running the test suites provided by RTAI on the test-bed CPU. The following results were obtained:

6.1. LATENCY TEST RESULTS

Latency is the time delay between occurrence of an event in a system and servicing of that event by the system. The latency test suite gives a measure of the average latency of the system under test. The test subjects the CPU to heavy loads by running complex algorithms involving intense calculations. The tests were conducted over a period of 1000 seconds. The latency test gives out results in form of table over a period with each row getting updated and presenting in detail the minimum and maximum latency readings along with overruns.

The test runs with the application loading RTAI modules for period of 100000 nanoseconds. The modules are dynamically linked and delinked at run time and the test starts giving out a measure of minimum latency, average latency and maximum latency on the screen. The test was initially conducted with default load period of 100000 nanoseconds for the RTAI modules used by the test application. For best performance the number of overruns should either be zero or should stay constant or increase very slowly and the latency timing results should not vary much with each update.

The screenshot of the test result is showed in Figure 6.1. The rows are the respective updates of the test suite in terms of maximum latency, average latency and number of overruns. The test suite performs the computations and the results are updated with the application running in one-shot mode of the RTAI kernel timer. The results were then plotted to present plots of average and maximum latency in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

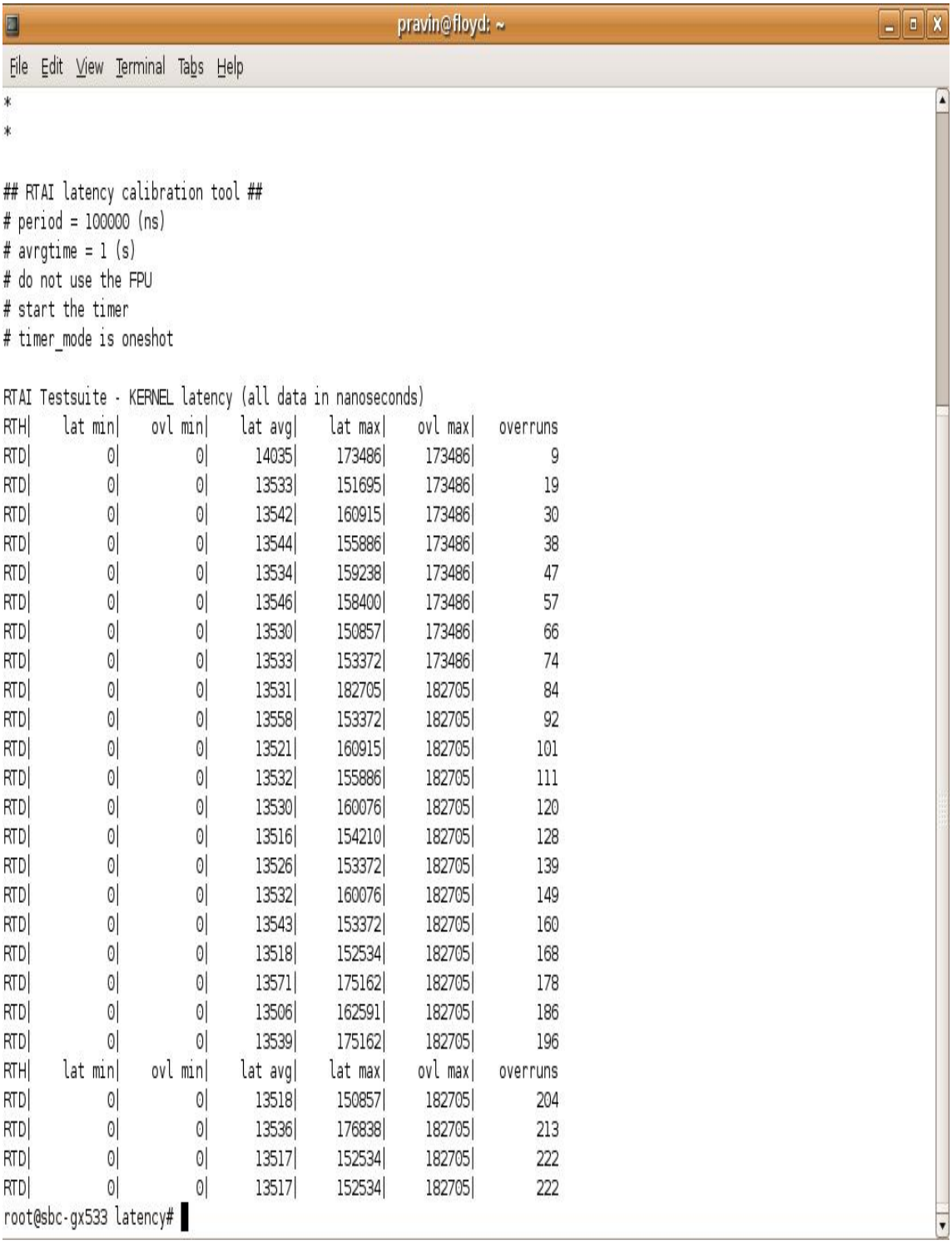

Figure 6.1: Latency test results with default period

Figure 6.2 shows the plot of the average latency with default period on the test-bed single board computer. The plot shows average latency wavering in the range of 13 to 15 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot in the application response in the initial update is observed where the average latency rises to 14.1 microseconds and then settles down to around 13.5 microseconds.

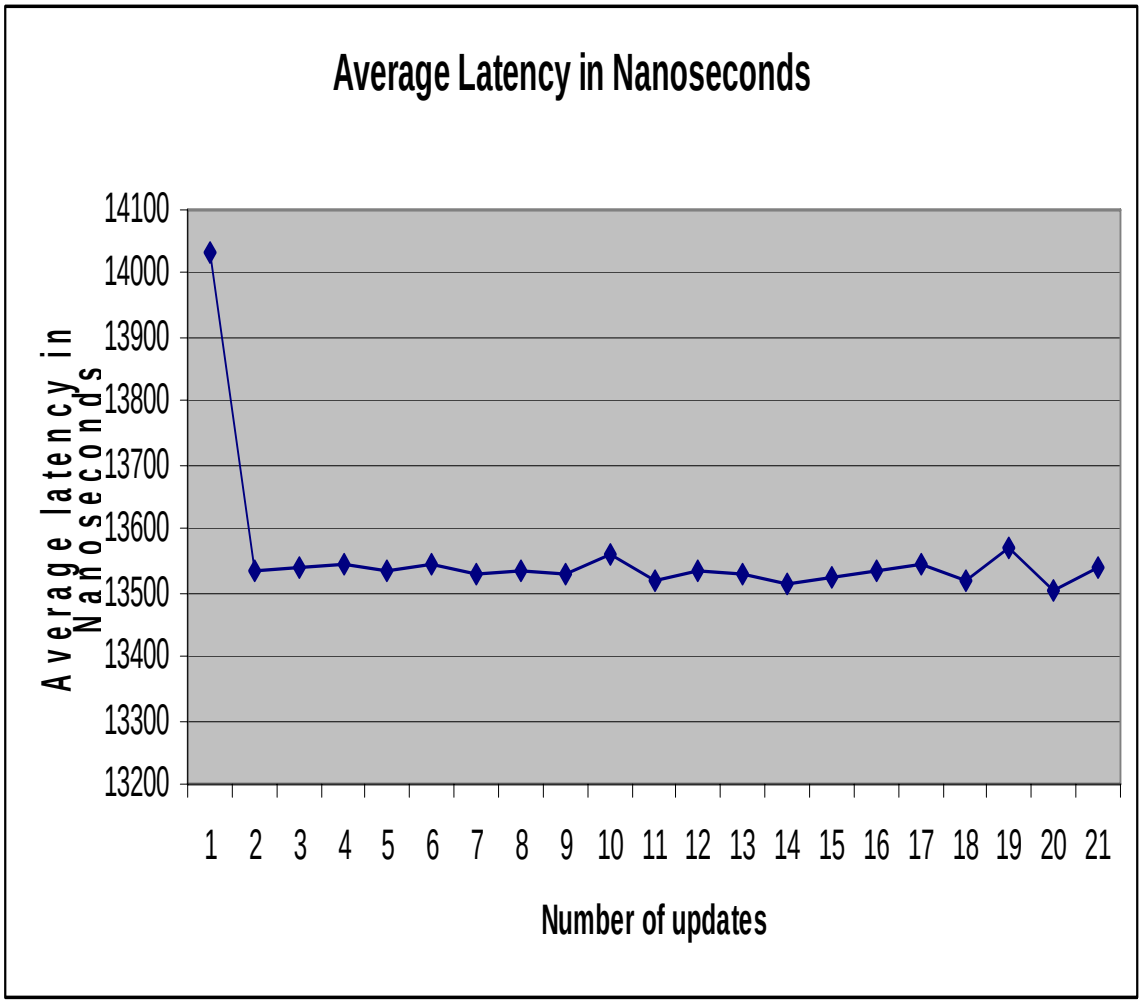

Figure 6.2: Plot of average latency for latency test suite with default period

Figure 6.3 shows the plot of the maximum latency with default period on the test-bed single board computer. The plot shows maximum latency wavering in the range of 140 to 180 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot is observed in the application response with maximum latency reaching around 180 microseconds and the maximum latency then settles down to around 160 microseconds.

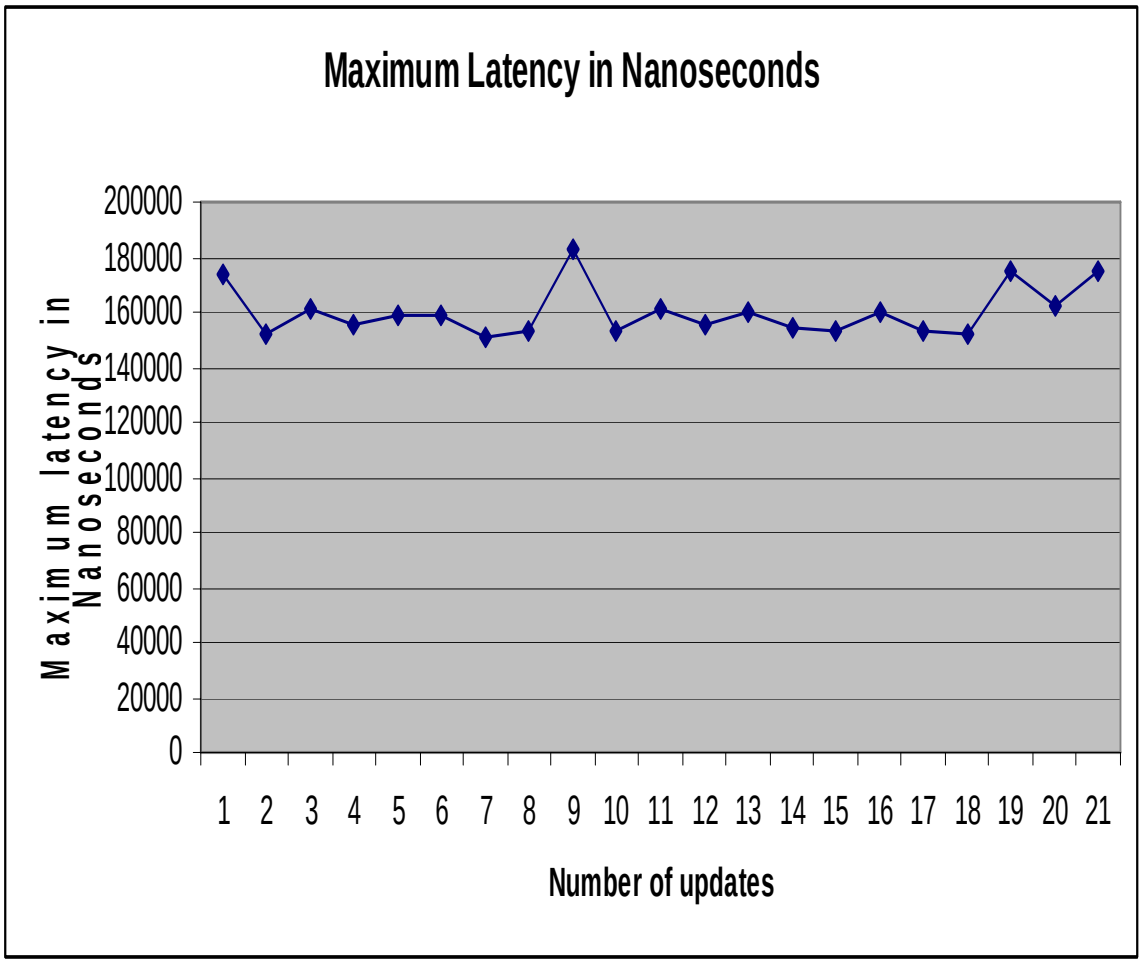

Figure 6.3: Plot of maximum latency for latency test suite with default period

The test indicated average latency in range of 13 to 14 microseconds and maximum latency in range of 170 to 180 microseconds. The results indicate good performance in terms of latency response of the system, however, the results obtained are not acceptable because the overruns should either be zero or should stay constant or increase very slowly.

The reason for this anomaly was because the test suite applications loaded the real time kernel modules for a very short period of 100,000 nanoseconds and thus the overruns increased constantly thereby the application kept on running in memory buffer overruns.

The solution to this problem was either to load the RTAI modules during booting of the test-bed single board computer or to implement a routine that should be called by the user during the start of his application.

The second approach was not practical enough since it could lead to calling of the routine repeatedly due to a programming error or abrupt stopping and re-executing of the application without rebooting the kernel. As a result it was decided that the first approach to make the RTAI modules load by default when the test-bed system kernel boots up.

To overcome this problem the required RTAI modules were configured to be loaded at boot time. In addition, the period of the modulethat was loaded during running of the test suites, was increased to 10,000,000 nano seconds. The modifications were made and it was ensured that the test-bed CPU loads the RTAI modules directly during boot time of the kernel.

Once the modifications were done, the tests were conducted again on the test-bed CPU. The test results were observed to be consistent in terms of zero number of overruns for each update and in terms of improved maximum latency performancethat improved to the range of 20 to 30 microseconds.

The screenshot of the test result with the period of loading of the RTAI modules to 10,000,000 nanoseconds is showed in Figure 6.4. The results were then plotted to present plots of average and maximum latency in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

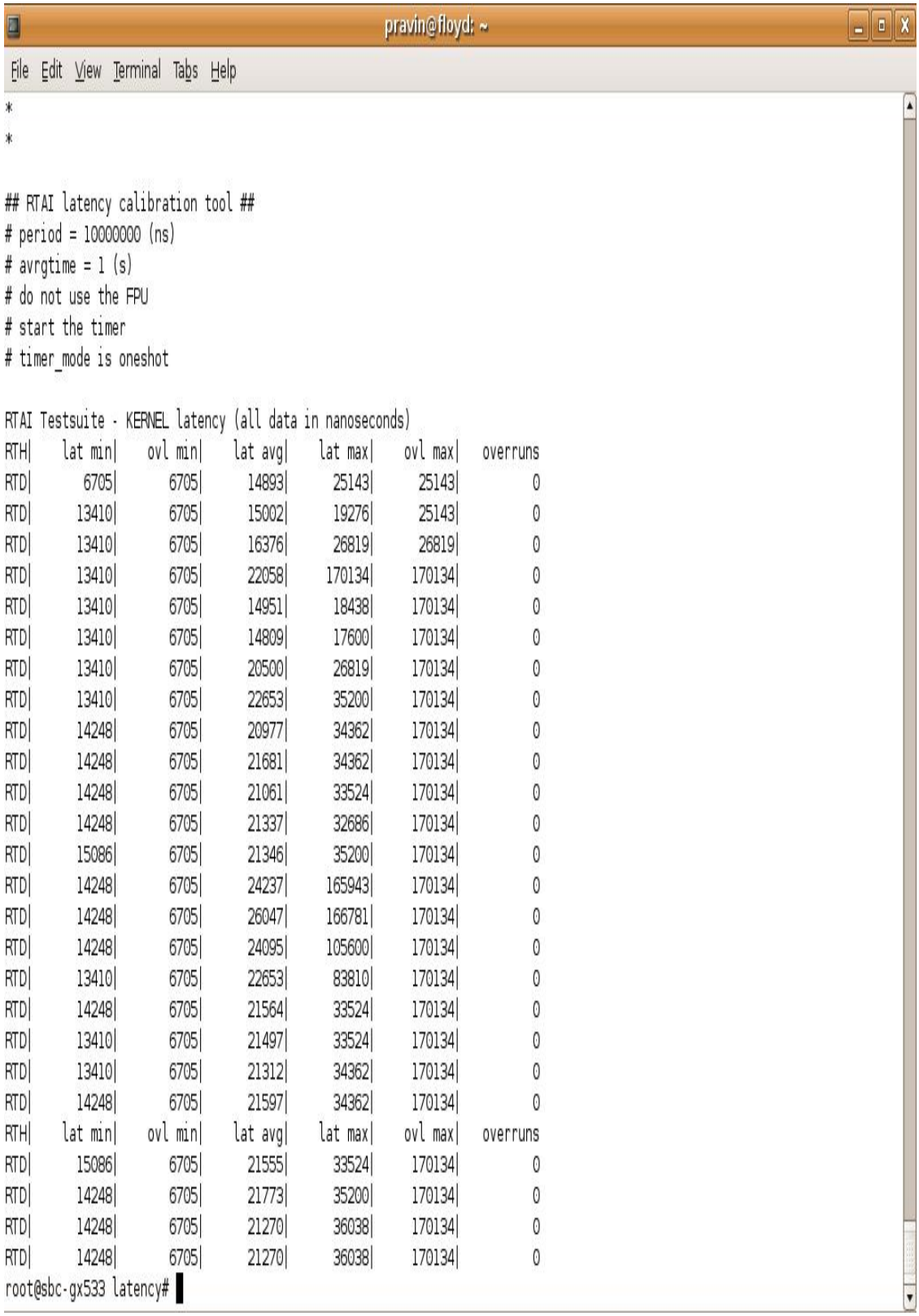

Figure 6.4: Latency test results with period increased to 10,000,000 nanoseconds

Figure 6.5 shows the plot of the average latency with period of 10,000,000 nanoseconds on the test-bed single board computer. The plot shows average latency wavering in the range of 15 to 20 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot in the application response is observed in the initial update where the average latency rises to 20 microseconds and then settles down to around 15 microseconds.

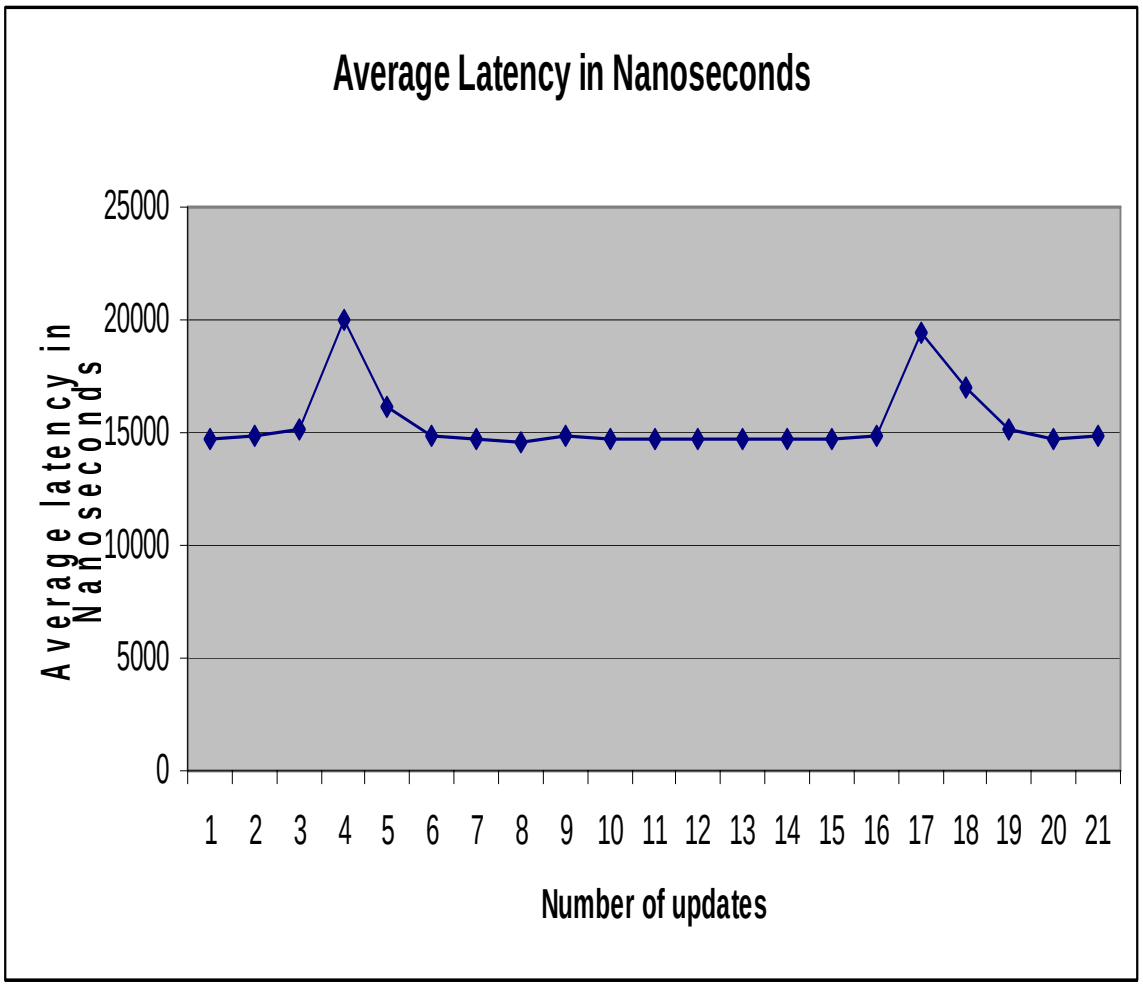

Figure 6.5: Plot of average latency for latency test suite with period of 10,000,000 ns

Figure 6.6 shows the plot of the maximum latency with period of 10000000 nanoseconds on the test-bed single board computer. The plot shows maximum latency wavering in the range of 200 to 400 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot in the application response is observed with maximum latency reaching around 400 microseconds and the maximum latency then settles down to around 200 microseconds.

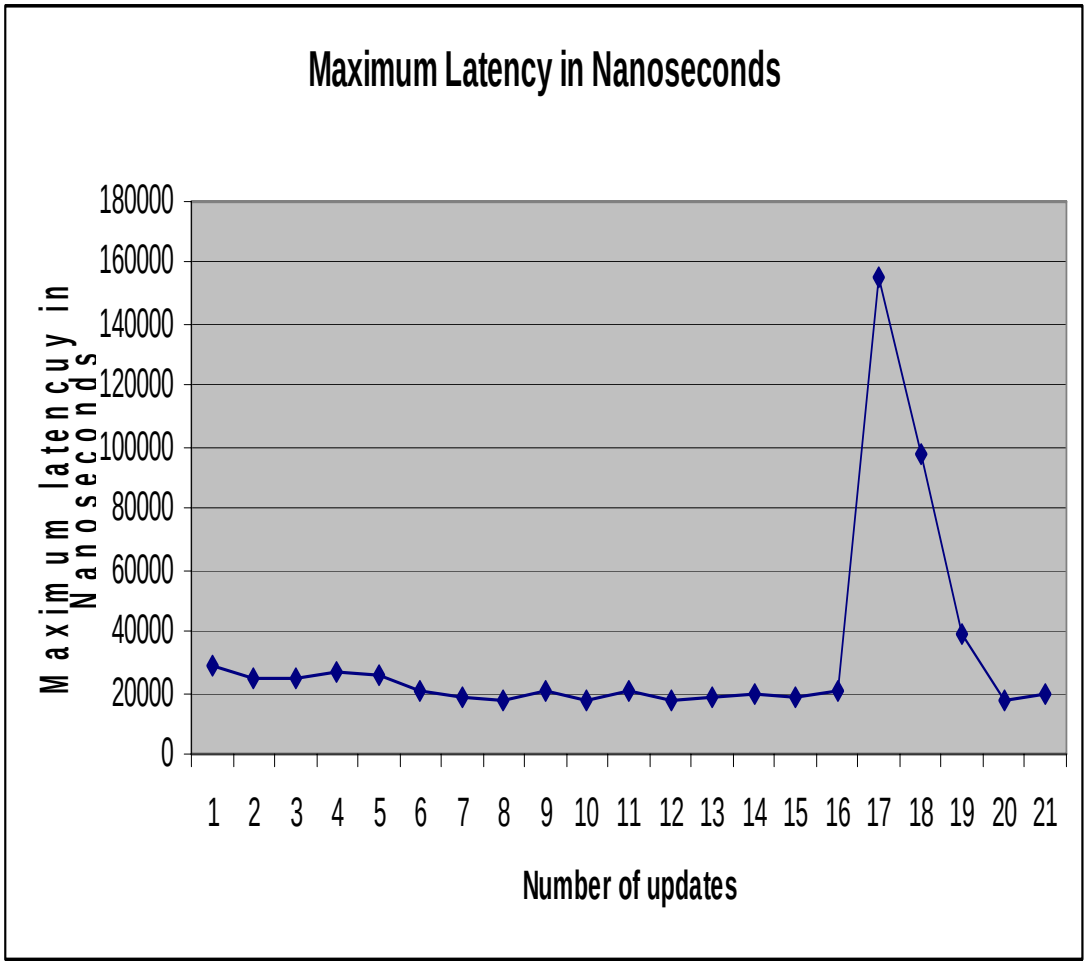

Figure 6.6: Plot of maximum latency for latency test suite with period of 10,000,000 ns

The test-bed system was then subjected to the latency test concurrently with an application that performed complex floating-point calculations in terms of finding square roots and cube roots of floating point numbers ranging from 1.5 to 1000.5 in increments of one in an infinite loop. The test served as stress test in terms of finding the test-bed system performance under demanding floating-point operations.

The test-bed system performance maintained its consistency in terms of latency in response to the concurrent running of the test suite with the floating point based application. The results strongly emphasize on the multitasking performance of the test-bed CPU indicating performance concurrent running of the real time threads declared in the test-bed software library.

The concurrent running of the real time threads is not as computationally demanding and intensive as compared to these tests. Hence, the consistency of performance of the real time application is concluded based on these tests.

The test-bed CPU does not have a separate floating-point unit. As a result, the floating-point application shares the processor along with other tasks. Since the floating-point tasks consume a large number of CPU cycles, the test-bed CPU is subjected to perform large amount of multitasking of concurrent applications and maintain the frequency of periodic real time threads.

The test-bed CPU gave successful results for maintaining its performance even with a computationally intensive task such as the floating-point application and the real time kernel proved its multitasking capabilities.

The floating-point application serves as an additional source of load on the RTAI scheduler and providing a good measure of the RTAI capabilities on the test-bed CPU. The test results are in accordance with [17] for test-bed equivalent configuration.

The screenshot of the test result running concurrently with an application involving floating point calculation is showed in Figure 6.7. The results were then plotted to present plots of average and maximum latency in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

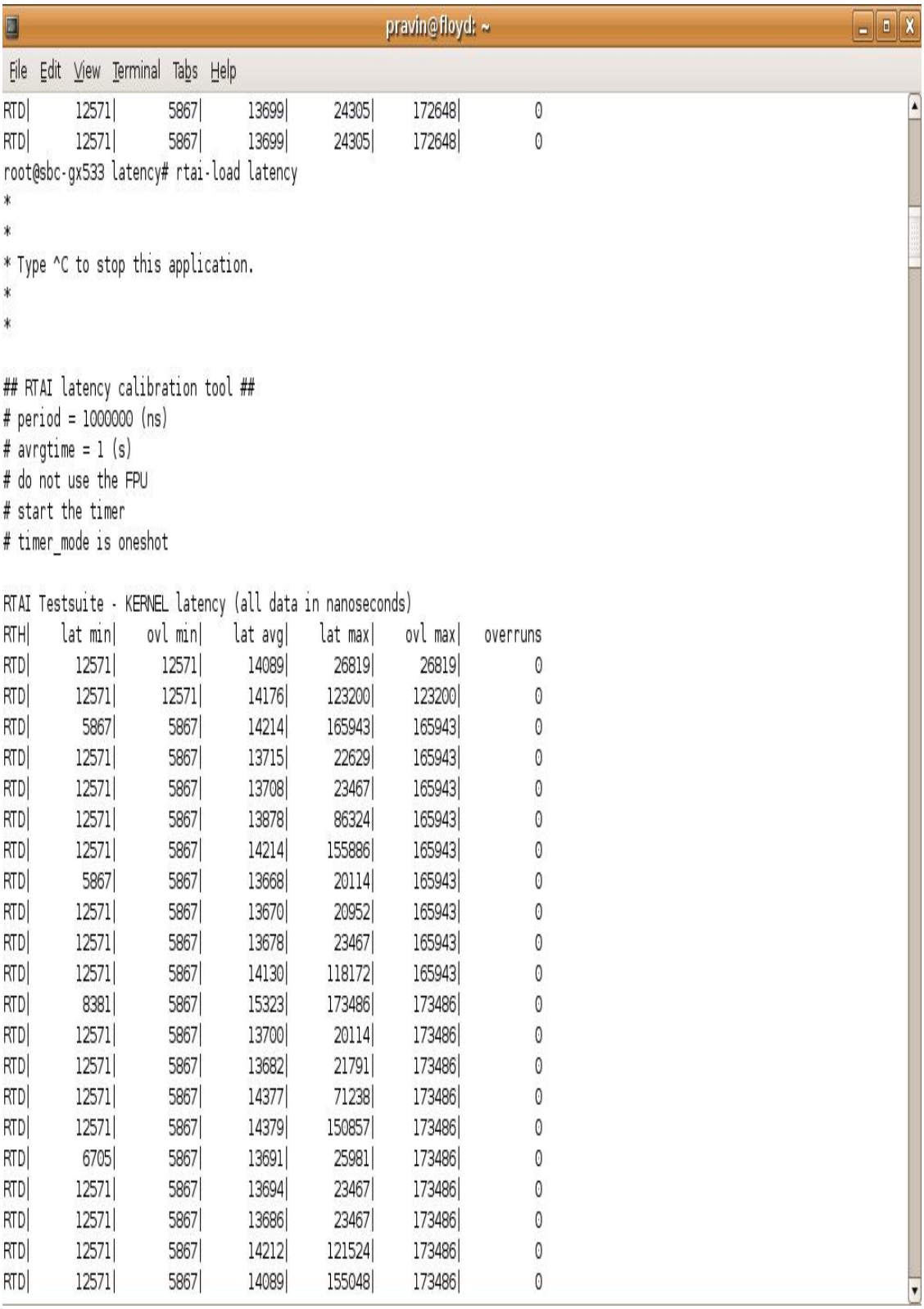

Figure 6.7: Latency test results with a concurrent floating-point application

Figure 6.8 shows the plot of the average latency with a concurrent floating-point application. The plot shows average latency wavering in the range of 13 to 14.5 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot in the response was observed where the average latency was as high as 14.5 microseconds and then settled down to around 13.5 microseconds.

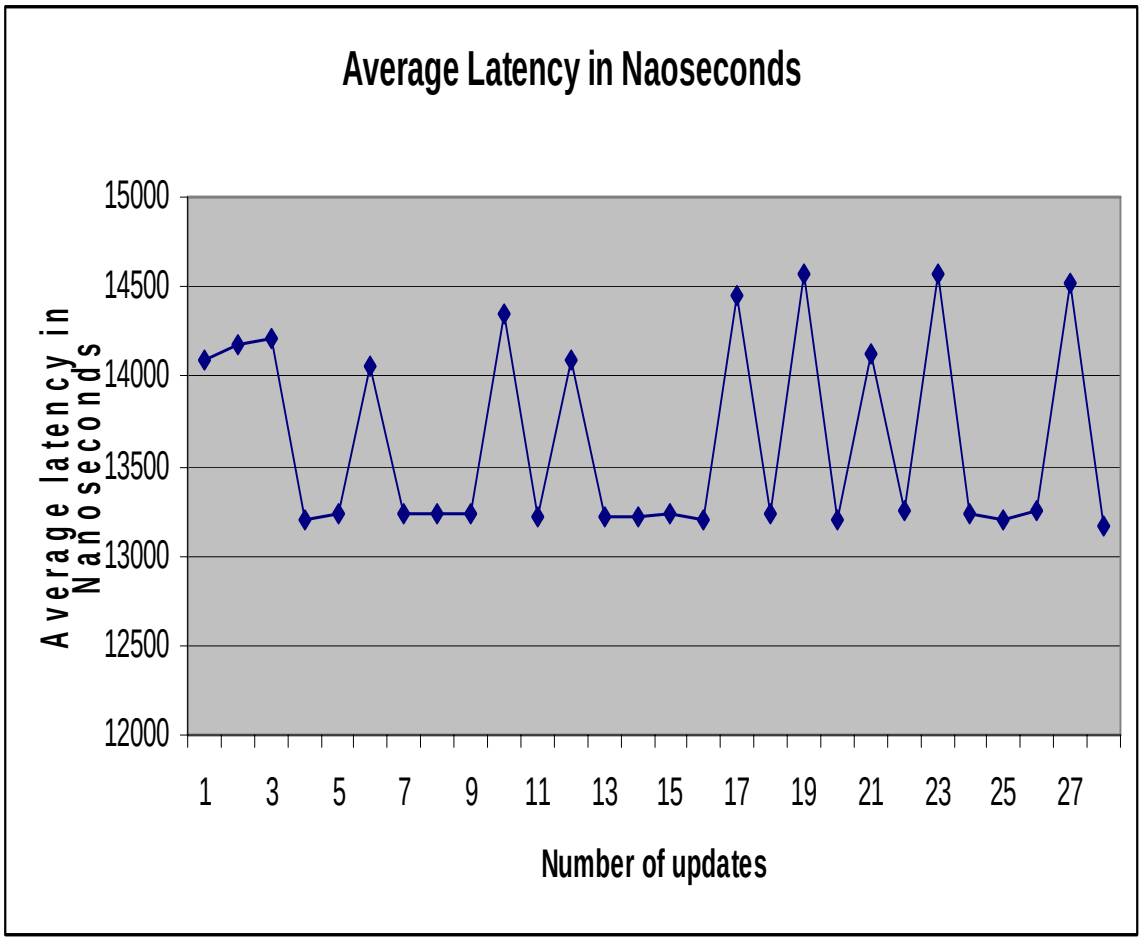

Figure 6.8: Plot of average latency for latency test suite with concurrent floating-point application

Figure 6.9 shows the plot of the maximum latency test result with a concurrent floating-point application. The plot shows maximum latency wavering in the range of 20 to 160 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. Occasional peaks of around 160 microseconds were observed which indicated the stress the test-bed CPU was put into owing to the floating-point application.

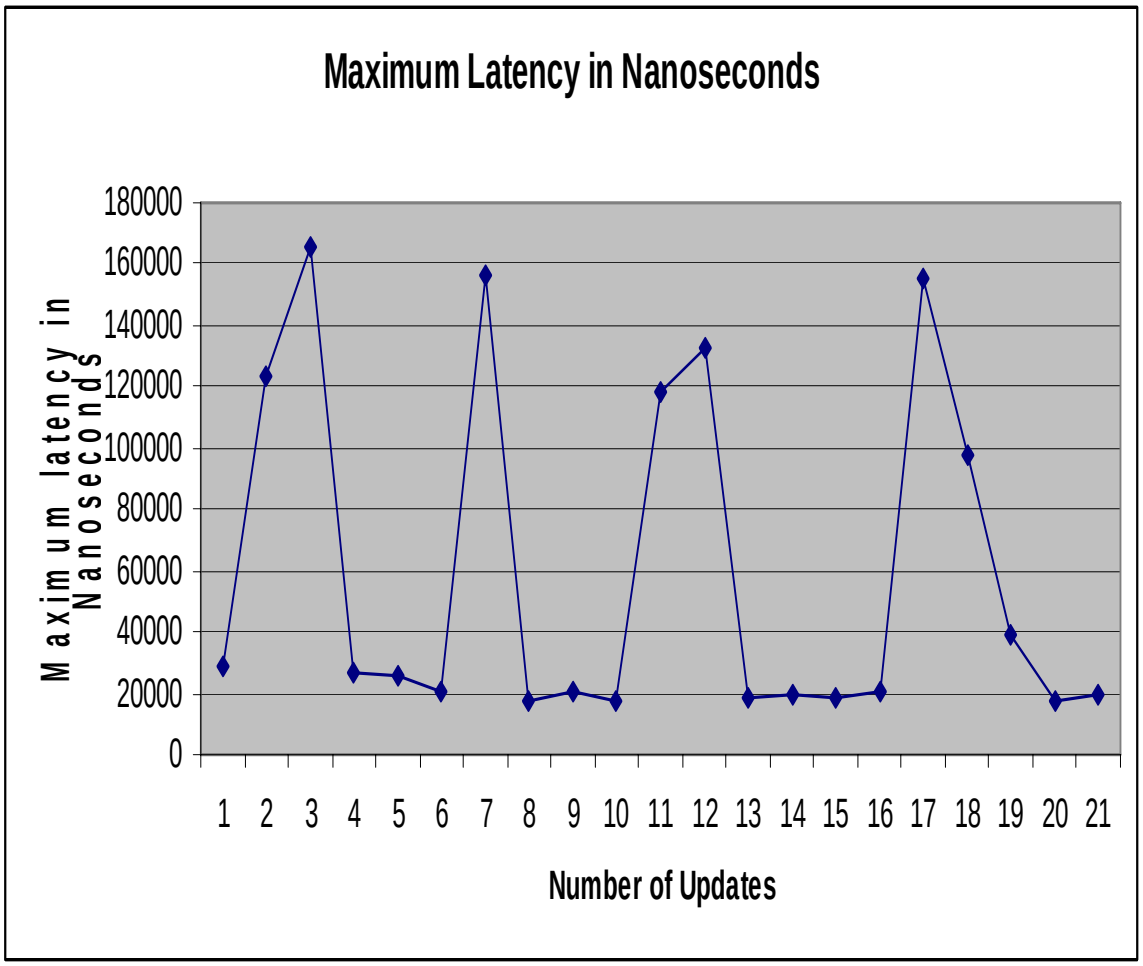

Figure 6.9: Plot of maximum latency for latency test suite with concurrent floating-point application

The test suite for latency was then executed in parallel with the test suite for preemptionthat subjects the RTAI schedulers to intense load. Thus, parallel running of both the tests subjects the test-bed CPU kernel to maximum load in terms of large number of interrupts provided by the test suite for latency and maximum scheduling demand through the test suite for preemption.

The preemption test and latency test run in one-shot mode to measure the difference in time between the expected switch time and the time when a task is actually called by the scheduler. However, in periodic mode of RTAI application the variation of the task period is used as measures of the scheduling latency and jitter. This is because the timer interrupt for RTAI is based on a time baseline that is different from the one used to carry out measurement calculations in the preemption and latency tests. As long as there is no major loss of timer interrupt, there will be no drift in periodic mode. Hence, the measures of this are valid measures.

The test-bed performed consistently under the intense load of the preemption and latency test suite in terms of average latency and maximum latency with a slight deterioration in the performance. The average latency was in range of 14 microseconds to 20 microseconds and maximum latency 28 microseconds to 35 microseconds.

This performance indicates the test-bed CPU performance in the worst case in case of an application having high frequency interrupts and scheduling demands in handling interrupts (hardware and software). The real time concurrent threads act as high priority software interrupts. The real time kernel challenge lies in handling the real time software interrupts under intense load conditions. The real time scheduler has to ensure the consistency of the performance for the real time functions and accordingly has to schedule the tasks and yet maintain the timing specification requirements. The RTAI kernel in combination with the test-bed CPU was able to effectively handle the specification demands of any real time application, which is evident from the test results.

The screenshot of the latency test result running concurrently with preemption test suite is showed in Figure 6.10. The results were then plotted to present plots of average and maximum latency in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

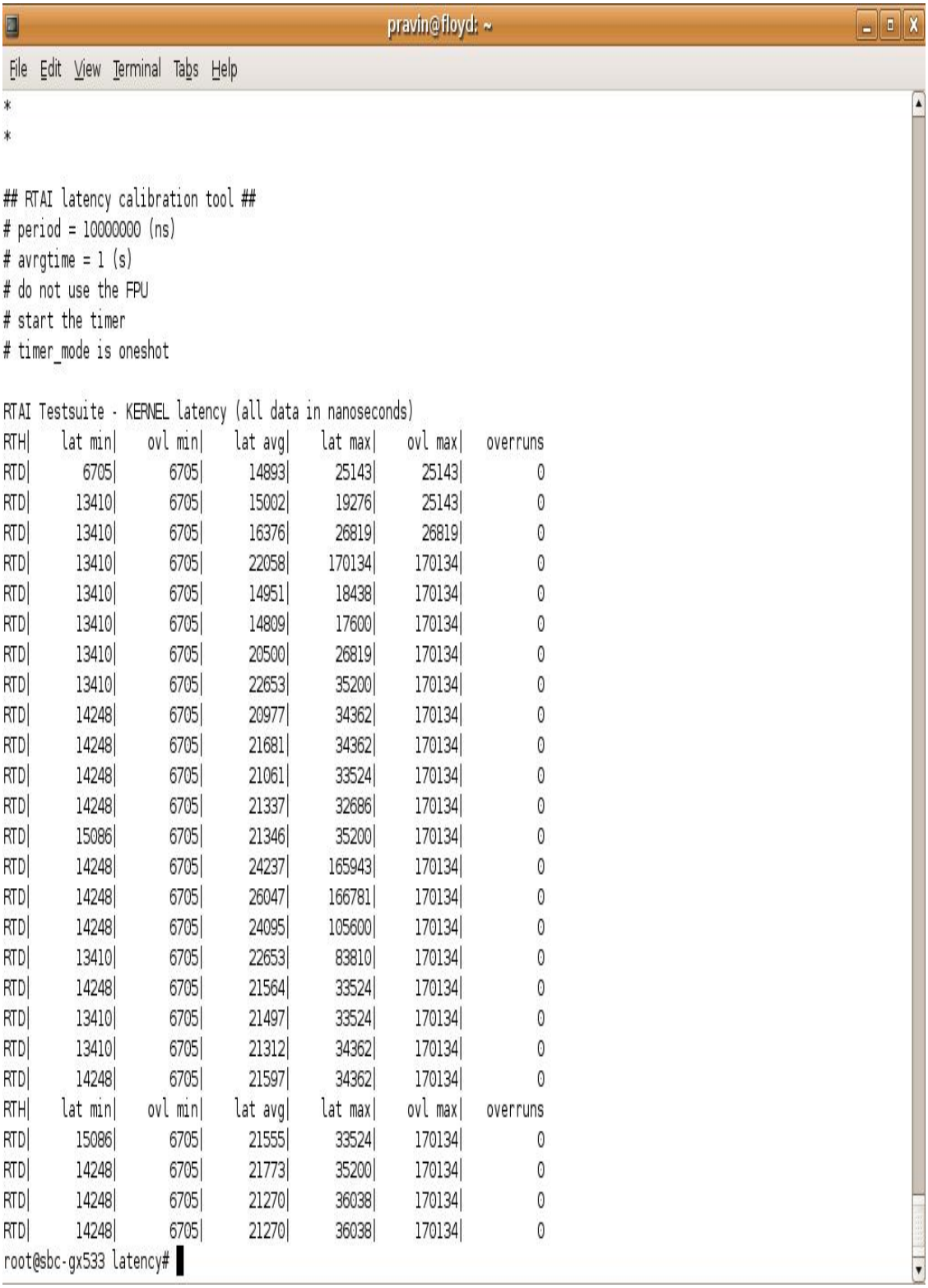

Figure 6.10: Latency test results with preemption test suite in parallel

Figure 6.11 shows the plot of the average latency for latency test suite with preemption test suite running in parallel. The plot shows average latency wavering in the range of 15 to 25 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. A minor shoot in the response was observed where the average latency was as high as 25 microseconds and then settled down to around 20 microseconds.

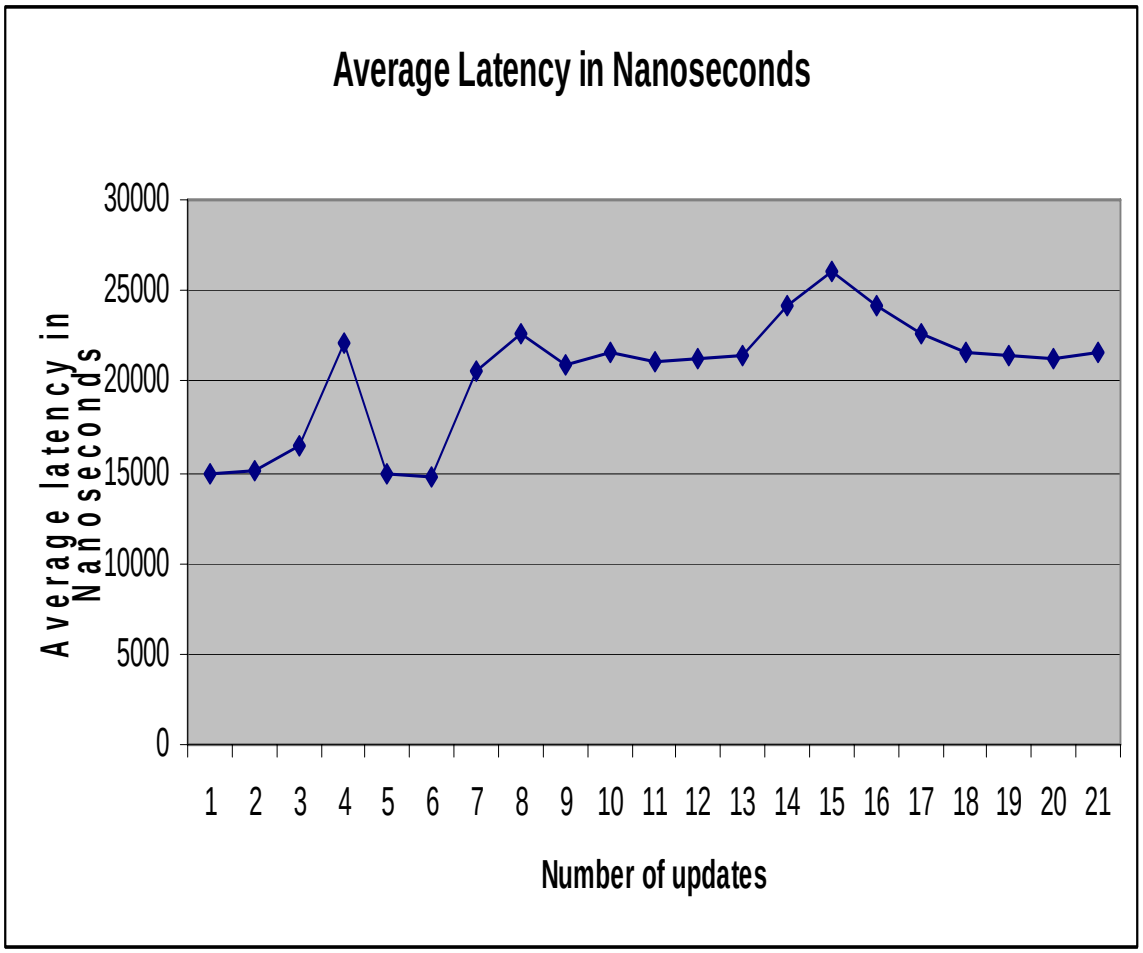

Figure 6.11: Plot of average latency for latency test suite with preemption test suite in parallel

Figure 6.12 shows the plot of the maximum latency test result with preemption test suite running in parallel. The plot shows maximum latency wavering in the range of 20 to 160 microseconds.

The plot presents average latency in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. Occasional peaks of around 1600 microseconds were observed which indicated the stress the test-bed CPU was put into owing to the parallel execution of the preemption test suite.

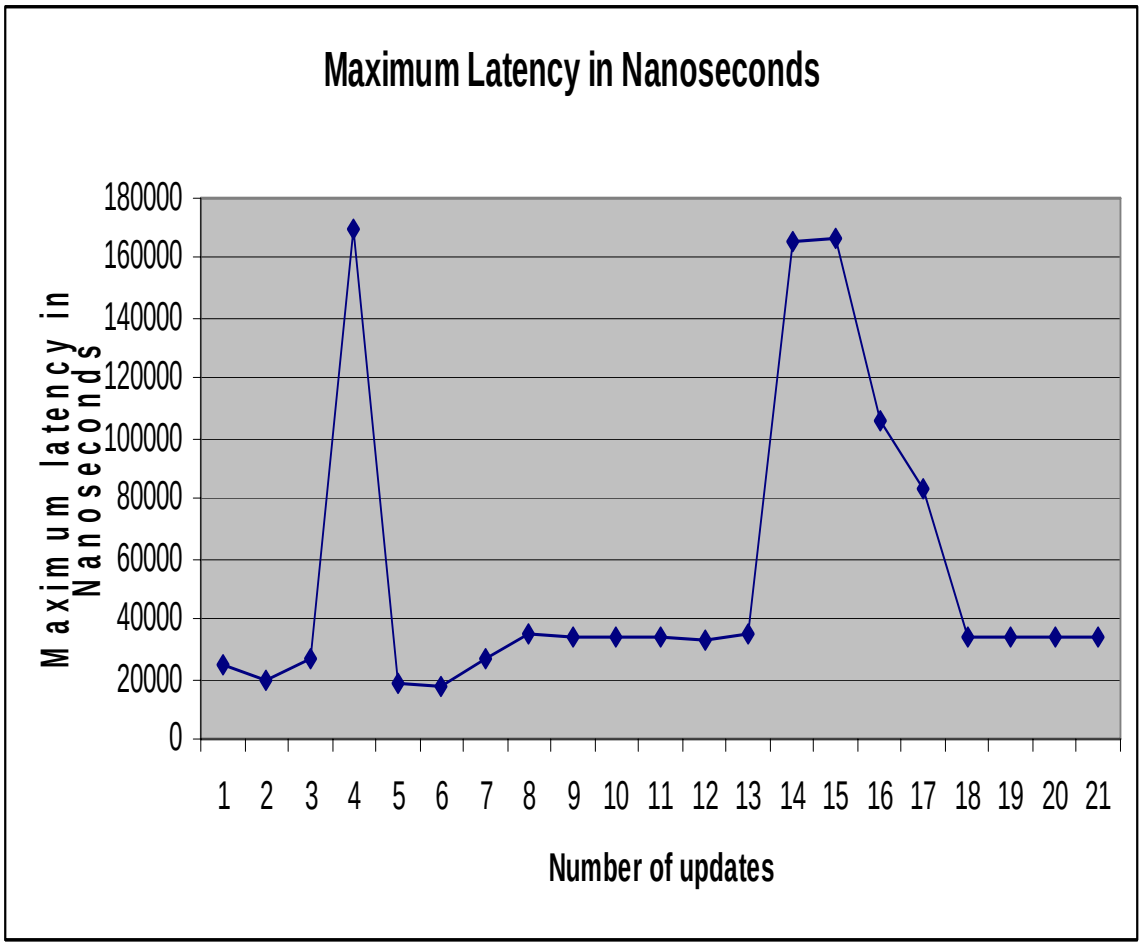

Figure 6.12: Plot of maximum latency for latency test suite with preemption test suite in parallel

6.2. PREEMPTION TEST RESULTS

Preemption means a high priority task preempts a low priority task. The preemption test is designed to verify the RTAI schedulers under intense load and provides as a stress utility test. The test suite combines the latency calibration task (highest priority task) with a fast task (second highest priority) and slow task (lowest priority task) and gives jitter results over execution of this test suite [17].

The test results are verified by launching the display utilitythat shows the tabulated results of minimum, maximum and average jitters of the latency check task, next to high priority fast task, and of the slow task, at the lowest priority. Reasonable jitters are a clear indication of preemption. The preempt test on CPU subjects the CPU up to the 75% computing power. The parameters of this test were adjusted according to test-bed machine specifications to avoid unfair locking of the scheduler.

Jitter in regards to real time application means the fluctuation in latency and response time of the system. Figure 6.13 shows summary of results obtained for the preemption test when the test was conducted over a period of 1000 seconds. The results show reasonable jitters indicating preemption.

Determination of Preemption is important because this test determines the time spent by test-bed single board computer when the high priority real time threads would preempt the low priority control task that would have access to CPU during the one millisecond sleep time of the threads. This result concludes that for periodic application reasonable preemption time performance was displayed by the test-bed single board computer and gives a good measure of the test-bed performance when the low priority user application would run concurrently with the high priority threads for sensor updates and the high priority actuator control functions defined in the test-bed software library.

The screenshot of the preemption test is presented in Figure 6.13. The results were then plotted to present plots of jitter time for fast task and jitter time for slow task in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

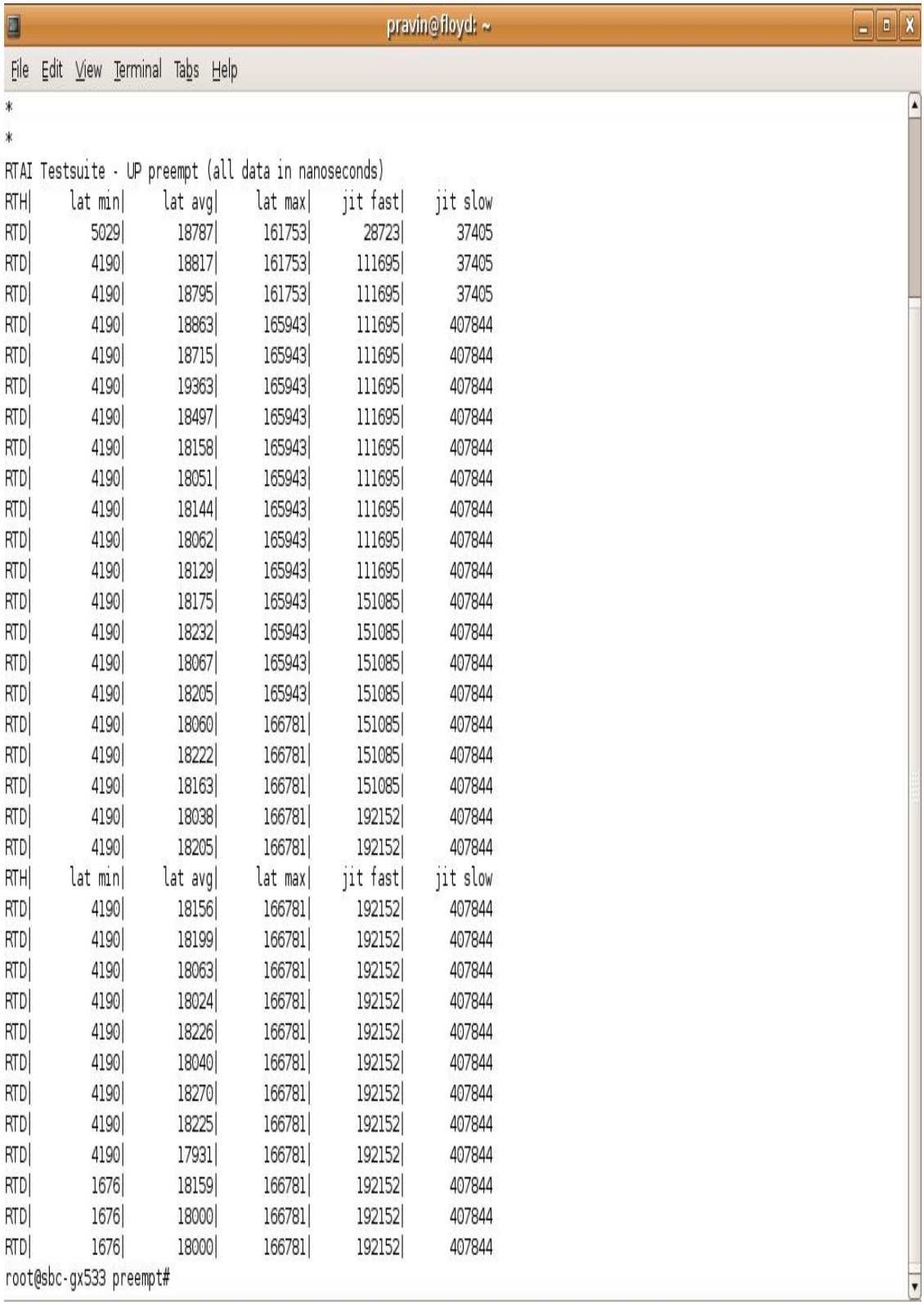

Figure 6.13: Preemption test results

Figure 6.14 shows the plot of the jitter for the fast task switching. The plot shows the jitter for fast task wavering in the range of 100 microseconds to 200 microseconds.

The plot presents jitter for fast task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates started with less than 50 microseconds of jitter time and then maintained the jitter at around 100 microseconds. As the updates progressed, the jitter was around 200 microseconds.

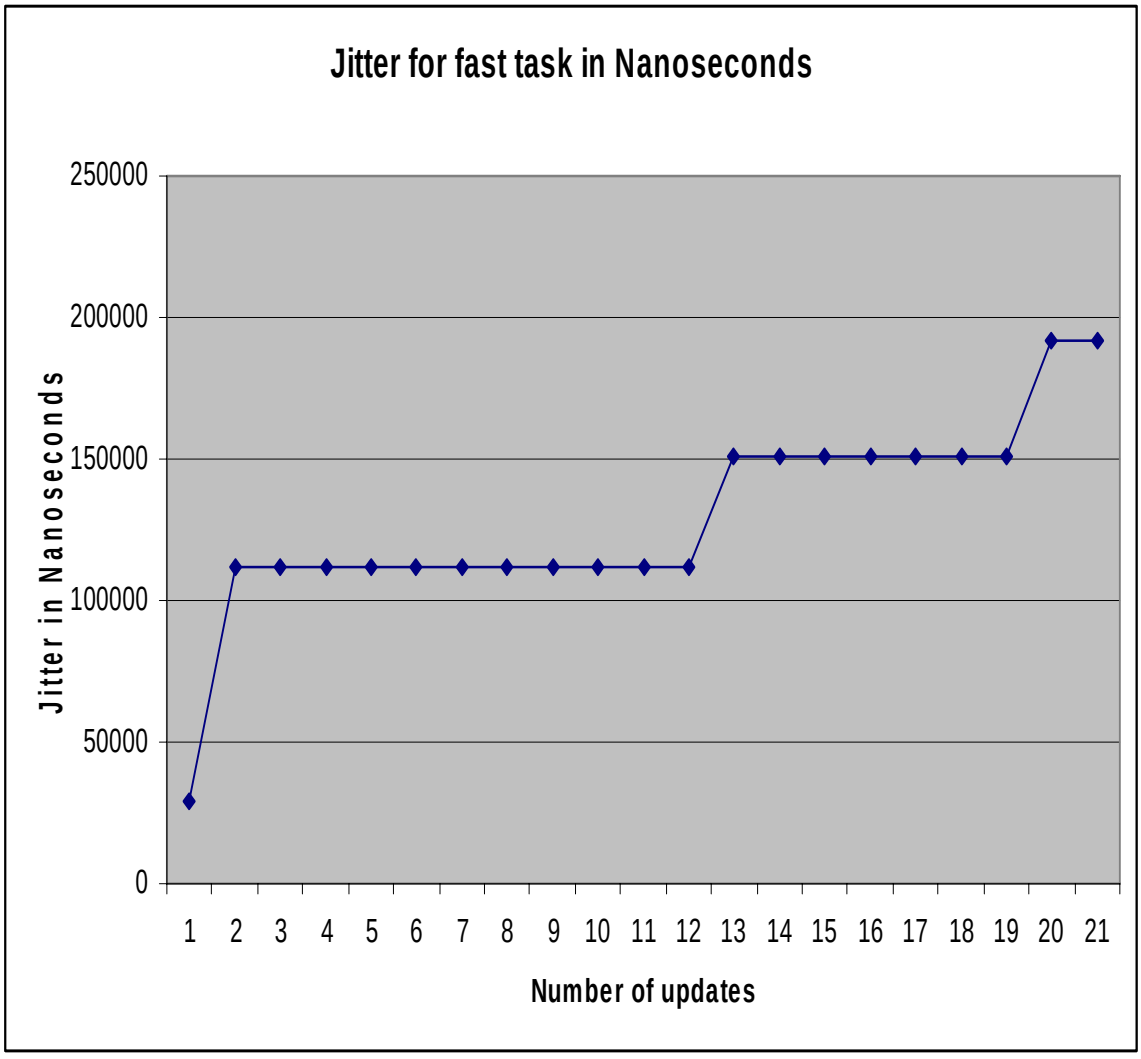

Figure 6.14: Plot of jitter for fast task

Figure 6.15 shows the plot of the jitter for the slow task switching. The plot shows the jitter for fast task wavering in the range of 100 microseconds to 200 microseconds.

The plot presents jitter for fast task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates maintained the jitter at around 100 microseconds and then as the updates progressed the jitter was around 200 microseconds.

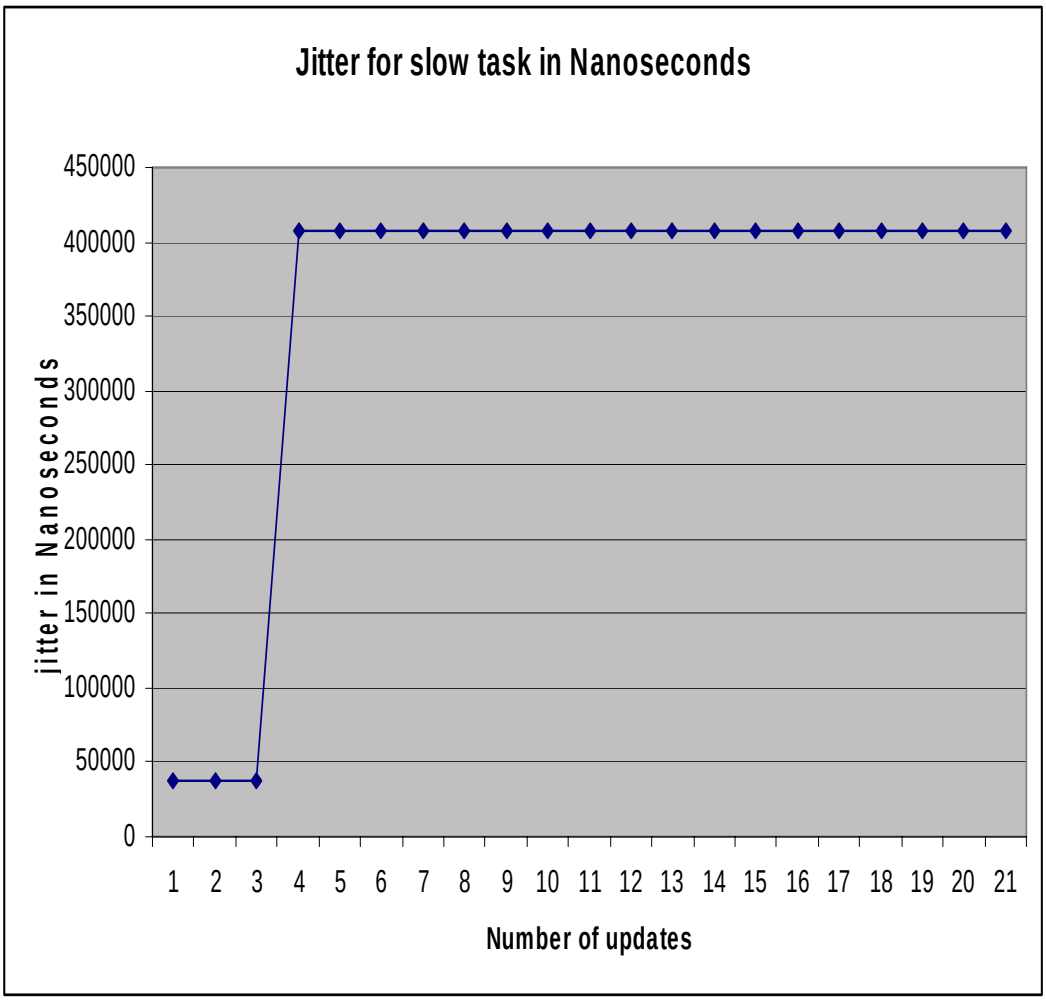

Figure 6.15: Plot of jitter for slow task

The test-bed system was then subjected to the preemption test concurrently with an application that performed complex floating-point calculations in terms of finding square roots and cube roots of floating point numbers ranging from 1.5 to 1000.5 in increments of one in an infinite loop. The test served as stress test in terms of finding the test-bed system performance under demanding floating-point operations.

The test-bed CPU does not have a separate floating-point unit. As a result, the floating-point application shares the processor along with other tasks. Since the floating-point tasks consume a large number of CPU cycles, the test-bed CPU is subjected to perform large amount of multitasking of concurrent applications and maintain the frequency of periodic real time threads.

The test-bed system performance maintained its consistency in terms of latency and jitter response. The results strongly emphasize on the multitasking performance of the test-bed CPU and indicate performance of the concurrent real time threads declared in the test-bed software library with periodic rate of one millisecond with a computationally intensive task running in parallel. The concurrent running of the real time threads is not as computationally demanding and intensive as compared to these tests. Hence, the consistency of performance of the real time application is concluded based on these tests.

The floating-point application serves as an additional source of load on the RTAI scheduler and providing a good measure of the RTAI capabilities on the test-bed CPU. The test results are in accordance with [17] for test-bed equivalent configuration. The test results presented reasonable jitter even under tough load conditions and indicated preemption even under the stress presented by the concurrent floating-point application.

The screenshot of the test result running concurrently with an application involving floating point calculation is showed in Figure 6.16. The results were then plotted to present plots of jitter time for fast task and jitter time for slow task in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

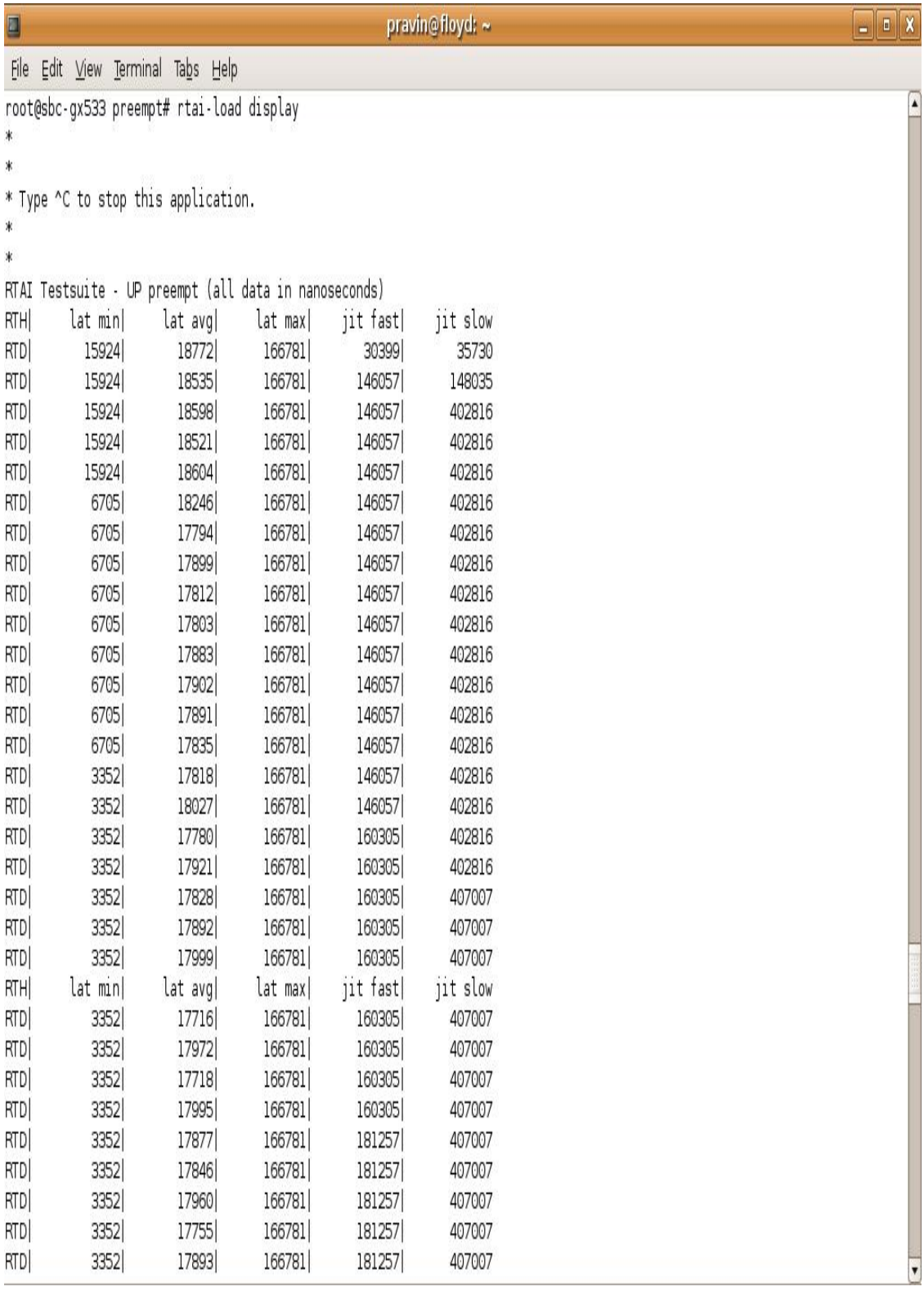

Figure 6.16: Preemption test results with a concurrent running floating-point application

Figure 6.17 shows the plot of the jitter for fast task with concurrent running floating point application. The plot shows the jitter for fast task wavering in the range of 200 microseconds to 1600 microseconds.

The plot presents jitter for fast task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates started with less than 400 microseconds of jitter time and then maintained the jitter at around 1400 microseconds. As the updates progressed, the jitter was around 1600 microseconds.

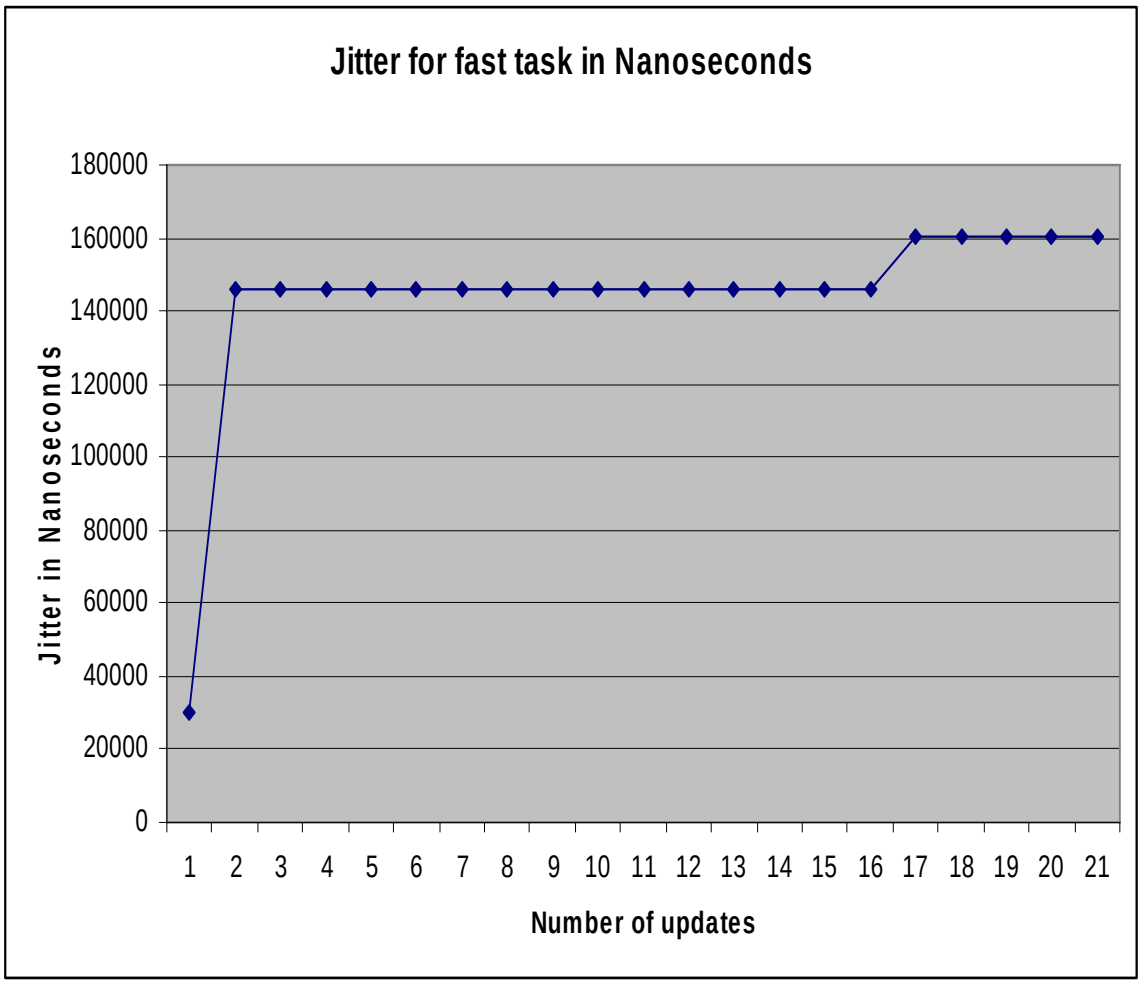

Figure 6.17: Plot of jitter for fast task with a concurrent running floating-point application

Figure 6.18 shows the plot of the jitter for slow task with a concurrent running floating point application. The plot shows the jitter for slow task wavering in the range of 400 microseconds to 450 microseconds.

The plot presents jitter for slow task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates started with less than 50 microseconds of jitter time indicating less initial demand on CPU and then maintained the jitter at around 400 microseconds consistently.

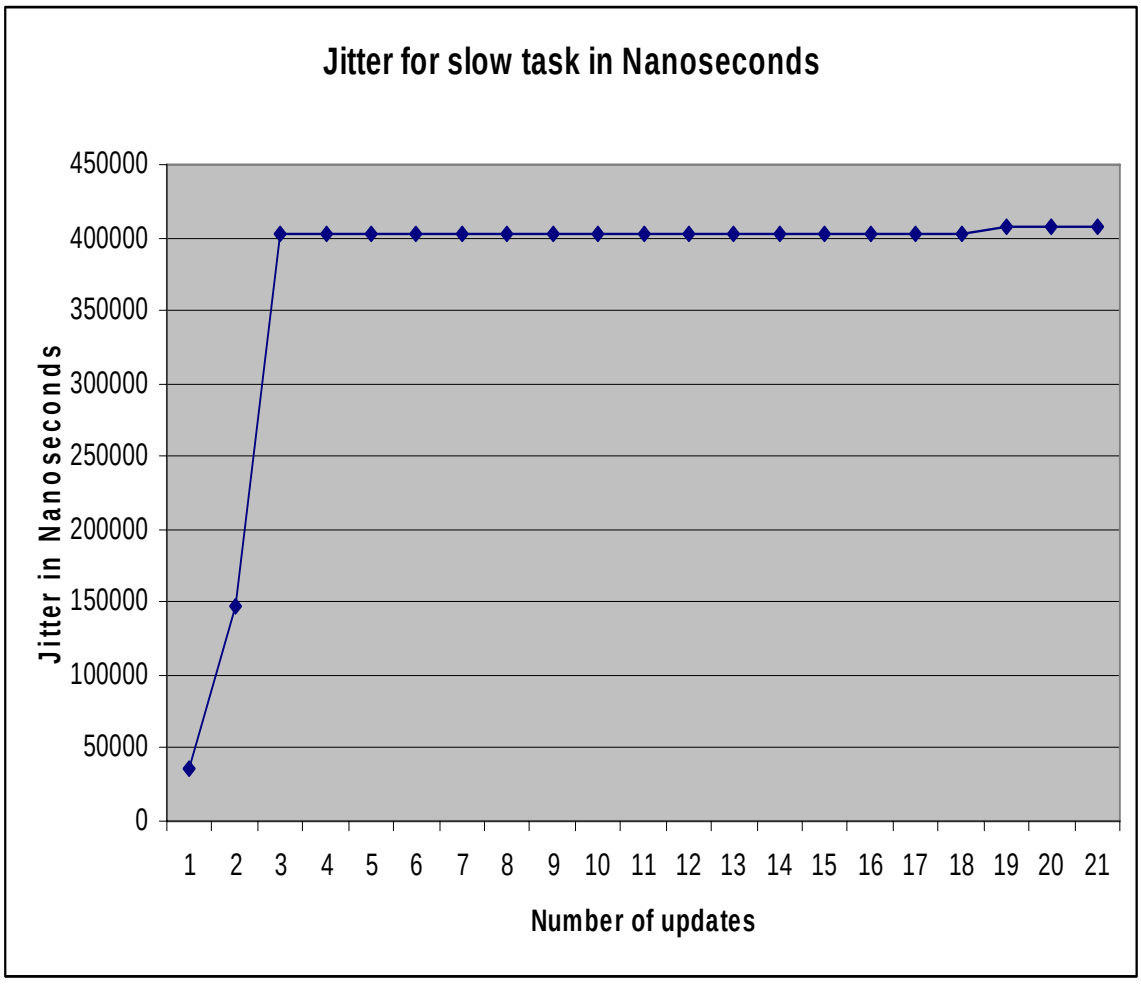

Figure 6.18: Plot of jitter for slow task with a concurrent running floating-point application

The test suite for preemption was then executed in parallel with the test suite for latencythat subjects the RTAI to handling maximum software interrupts at high frequency. Thus, parallel running of both the tests subjected the test-bed CPU kernel to maximum load in terms of large number of interrupts provided by the test suite for latency and maximum scheduling demand through the test suite for preemption.

Since the preemption, jitter test creates a single real-time thread that asks to be awakened periodically, and compares expected wake time to actual wake time the test-bed CPU was subjected to heavy and varying loads in form of the latency test suite running in parallel. Since jitter is mostly result of bus and memory contention the latency test suite serves as perfect load and test bench for testing the jitter result of the test-bed because the latency test suite subjects the CPU to loads on bus and memory contention.

Minimum interrupt latency which is largely determined by the interrupt controller circuit and its configuration can also effect the jitter in the interrupt latencythat can drastically affect the real-time schedulability of the system. Maximum interrupt latency is largely determined by the methods an operating system uses for interrupt handling. Maximum latency in case of RTAI test suite is a measure of how well RTAI handles maximum load in terms of large number of interrupts and maximum scheduling demand placed by an application on the CPU.

The test-bed performed consistently under the intense load of the preemption and latency test suite in terms of average latency and maximum latency with a slight deterioration in the performance. The jitter for fast task stayed consistently below 200 microseconds and the jitter for slow task stayed in range of 400 microseconds.

This performance indicates the test-bed CPU performance in the worst case in case of an application having high frequency interrupts and scheduling demands in handling the high priority real time threads.

The screenshot of the preemption test result running concurrently with latency test suite is showed in Figure 6.19. The results were then plotted to present plots of jitter time for fast task and jitter time for slow task in nanoseconds against the number of updates thereby presenting a graphical representation of the test results.

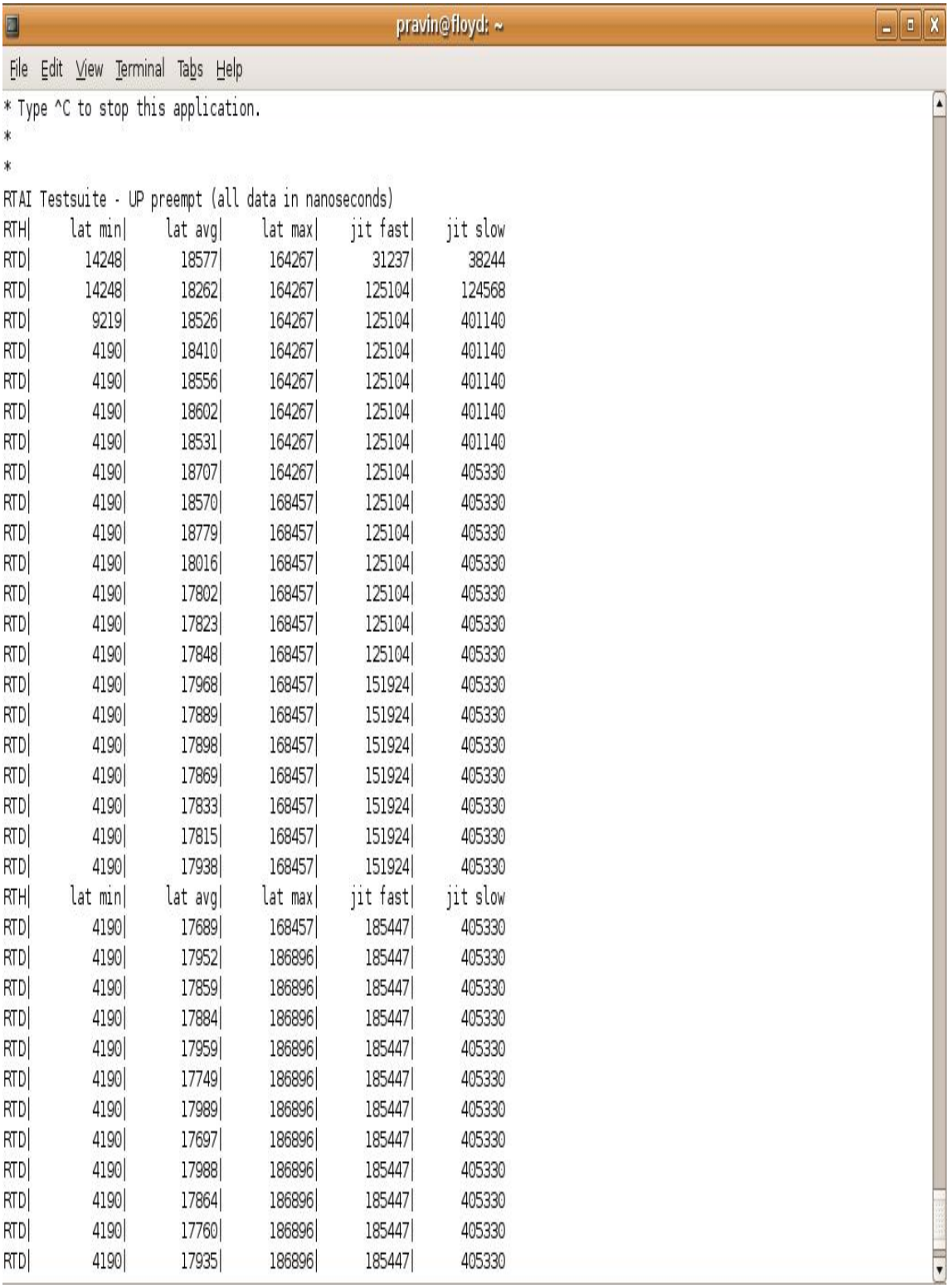

Figure 6.19: Preemption test results with latency test running in parallel

Figure 6.20 shows the plot of the jitter for fast task with latency test suite running concurrently. The plot shows the jitter for fast task wavering in the range of 120 microseconds to 160 microseconds.

The plot presents jitter for fast task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates started with less than 400 microseconds of jitter time indicating less initial demand on CPU and then maintained the jitter at around 1200 microseconds. As the updates progressed, the jitter was around 1600 microseconds.

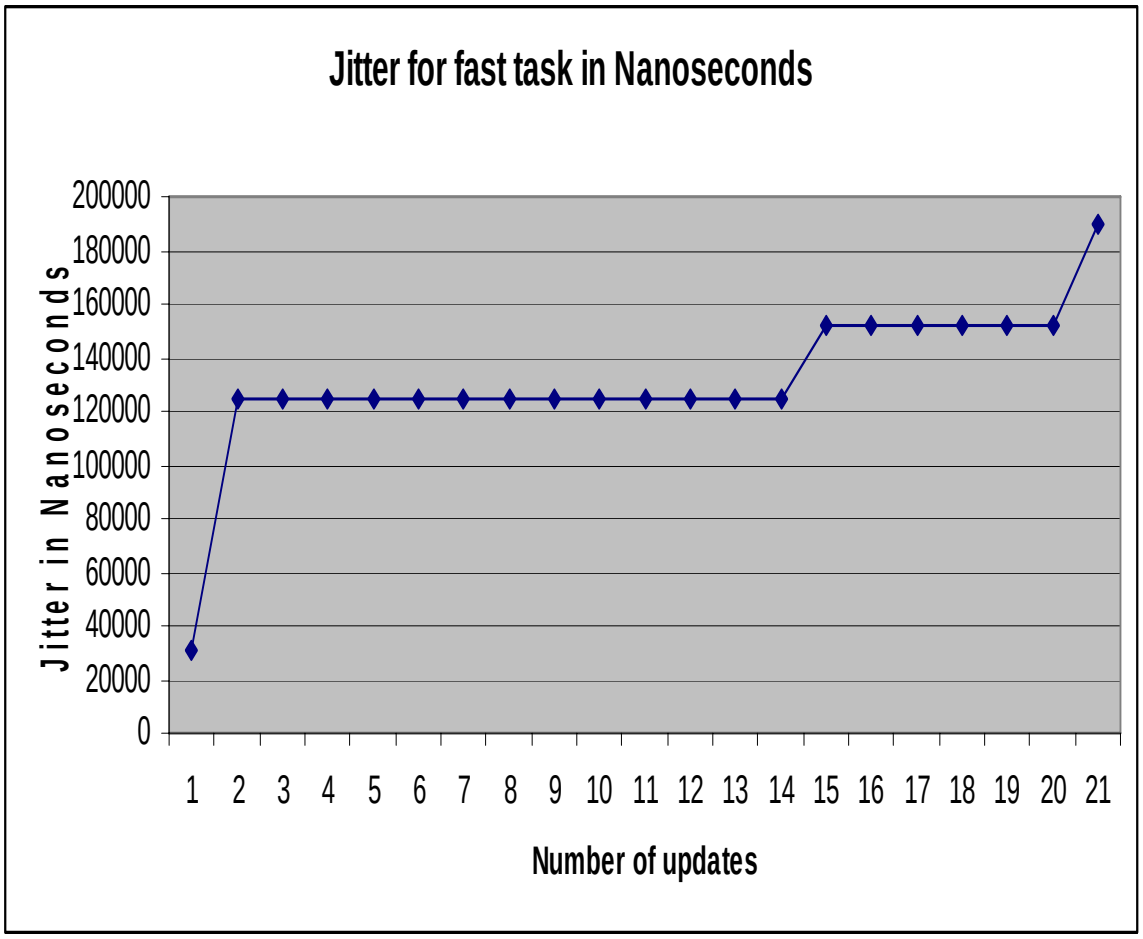

Figure 6.20: Plot of jitter for fast task with latency test running in parallel

Figure 6.21 shows the plot of the jitter for slow task with latency test suite running concurrently. The plot shows the jitter for slow task wavering in the range of 400 microseconds to 450 microseconds.

The plot presents jitter for slow task in nanosecond on the vertical axis and presents the number of updates on the horizontal axis. The initial updates started with less than 50 microseconds of jitter time indicating less initial demand on CPU and then maintained the jitter at around 400 microseconds consistently.

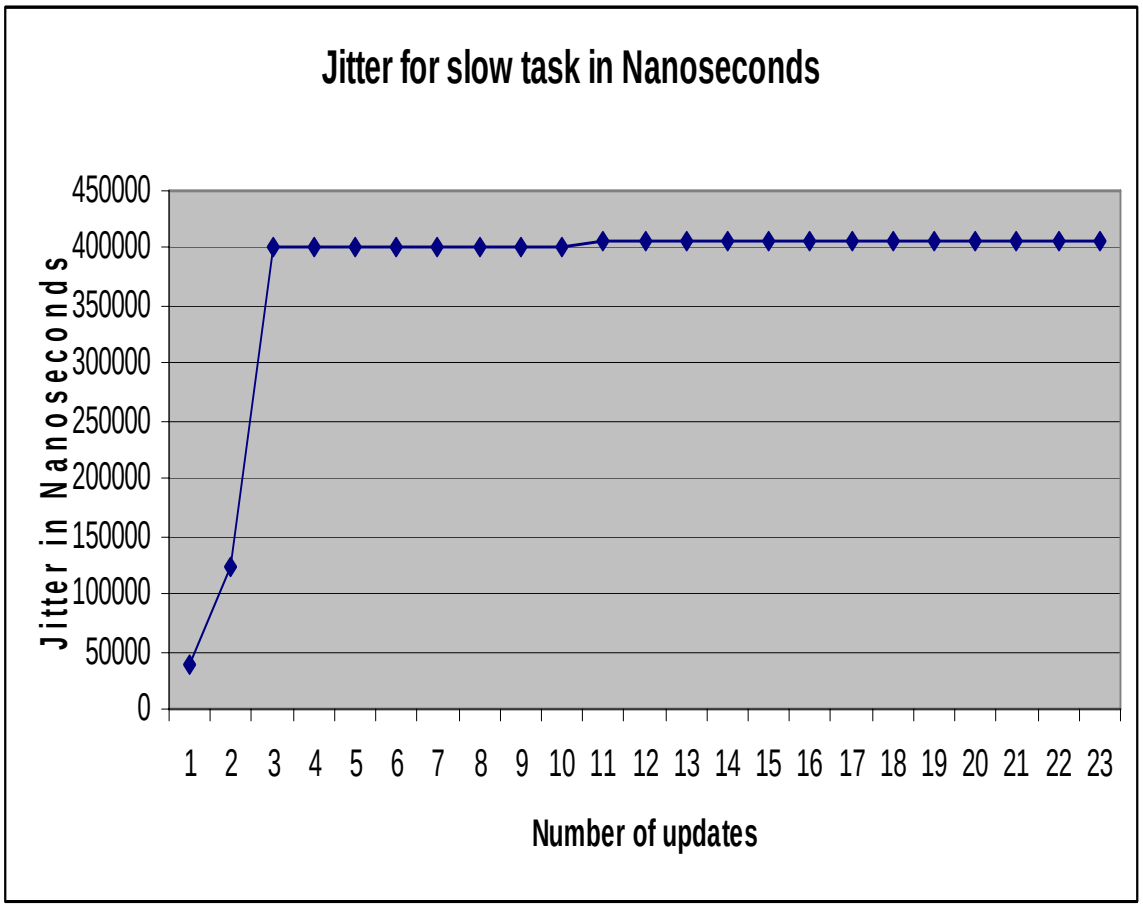

Figure 6.21: Plot of jitter for slow task with latency test running in parallel

6.3. SWITCHES TEST RESULTS

The switches test is a measure on context switching performance of the RTAI kernel on hardware its running. A context is the contents of a CPU's registers and program counter at any point in time. Context switching can be described in slightly more detail as the kernel performing the following activities with regard to processes (including threads) on the CPU: (1) suspending the progression of one process and storing the CPU's state (i.e., the context) for that process somewhere in memory, (2) retrieving the context of the next process from memory and restoring it in the CPU's registers and (3) returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process. A context switch is also sometimes described as the kernel suspending execution of one process on the CPU and resuming execution of some other process that had previously been suspended.

Switches test indicates maximum time required by RTAI for context switching [18] amongst tasks and disabling interrupts. To do so the test suite uses a repeated sequence of suspend/resume and semaphore signal/wait, with the FPU support and under a relatively heavy load of about 10 tasks and 40000 switches.

The test suite when executed gives out the performance of the system under test in terms of number of tasks, number of switches, time required to execute the tasks and the switching time. The switches test was conducted over ten times and after execution of each test, the result was printed on the screen. The screenshot of the switches test results is presented in Figure 6.22.

The switches test determined the context switching performance of the test-bed single board computer and gave a good summary of the worst-case time required by RTAI on test-bed to do context switching among the test-bed software library tasks and the user application tasks. The average switching time was in range of 5000 nanoseconds to 6000 nanoseconds. The switches test is importance in predicting the handling of test-bed CPU in handling the switching of the application tasks.

```
FOR 10 TASKS: TIME 212 (ms), SUSP/RES SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5302 (ns)

FOR 10 TASKS: TIME 215 (ms), SEM SIG/WAIT SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5386 (ns)

FOR 10 TASKS: TIME 233 (ms), RPC/RCV-RET SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5827 (ns)

FOR 10 TASKS: TIME 211 (ms), SUSP/RES SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5279 (ns)

FOR 10 TASKS: TIME 215 (ms), SEM SIG/WAIT SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5365 (ns)

FOR 10 TASKS: TIME 231 (ms), RPC/RCV-RET SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5772 (ns)

FOR 10 TASKS: TIME 212 (ms), SUSP/RES SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5302 (ns)

FOR 10 TASKS: TIME 215 (ms), SEM SIG/WAIT SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5367 (ns)

FOR 10 TASKS: TIME 232 (ms), RPC/RCV-RET SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5789 (ns)

FOR 10 TASKS: TIME 211 (ms), SUSP/RES SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5283 (ns)

FOR 10 TASKS: TIME 215 (ms), SEM SIG/WAIT SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5367 (ns)

FOR 10 TASKS: TIME 232 (ms), RPC/RCV-RET SWITCHES 40000, SWITCH TIME (INCLUDING FULL FP SUPPORT) 5812 (ns)
```

Figure 6.22: Switches test results

Figure 6.23 shows the plot of the switches test result. The plot shows the task switching time for 10 respective tests wavering in the range of 5.3 microseconds to 5.8 microseconds.

The plot presents task-switching time in nanosecond on the vertical axis and presents the respective test number on the horizontal axis. The test maintained consistent results for all the tests showing consistent switching time.

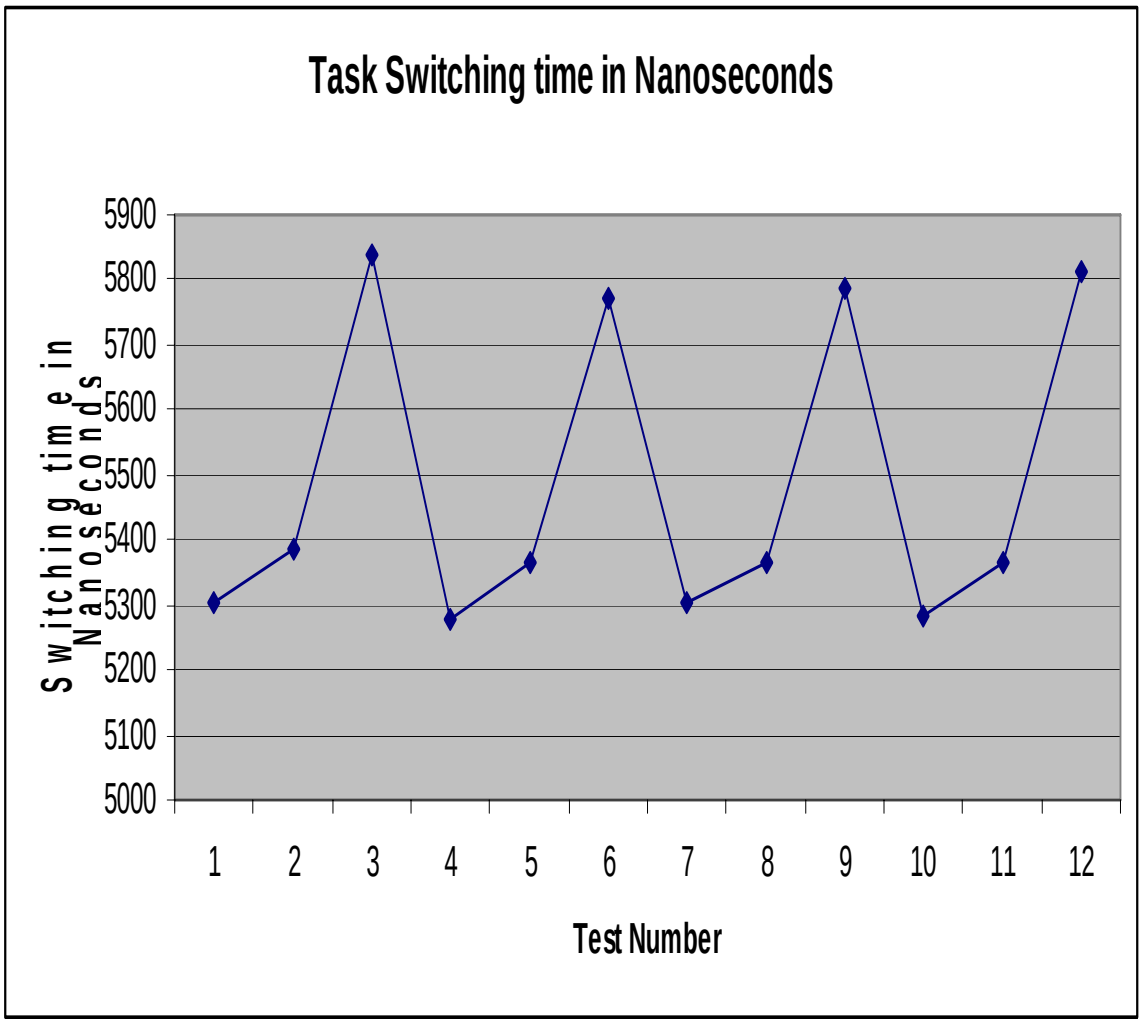

Figure 6.23: Plot of switches test results

7. CONCLUSIONS AND FUTURE WORK

The work presented in this thesis presents a real time approach in the development of test-bed for autonomous vehicle control schemes. Autonomous vehicle application developers can easily develop their applications based upon the test-bed software library. The real time implementation of the test-bed library ensures the sensor update and actuator control events are never missed, thus aiding development of an efficient system.

The test-bed library allows a user friendly API providing all the basic functions required for motion control in either position mode or velocity mode. The user of this test-bed has to develop application for Autonomous vehicle navigation steering control and steering angle generation and link it to the test-bed software library to use its API.

The test-bed library gave excellent consistent performance in terms of responding to the latency, preemption and switches test suite. The test-bed proved consistent in its performance when the test suites were executed in parallel with each other and with computationally intensive tasks. The test suite results are a good measure of how the test-bed will respond to the execution of the Autonomous Vehicle Navigation Application.

Hardware updates in future will improve the accuracy of testing autonomous vehicle applications on the test-bed. Fitting of the test-bed single bed computer with a rechargeable portable battery will add much needed mobility to the experiments. Right now, the test-bed single board computer runs on the dealer supplied wired battery. Adding a wireless network interface on the test-bed single board computer would add flexibility in remote debugging and remote testing of applications on the test-bed. Adding an H-bridge type power supply to the drive motor would add better motion control options.

Additional configuration of RTAI would further improve the performance of the test-bed real time kernel. [17] Lists plenty of methods of RTAI performance optimization. The test-bed library was developed and executes completely in user space thereby eliminating programming and debugging hassles. However, this approach costs some overhead in terms of latency, scheduling and task switching since the real time

application executes completely in user space. Next approach in this work would be to develop the library completely in kernel space and load the library in the test-bed as kernel module. The kernel module would then be linked and delinked dynamically as per the execution of the application.

The test-bed library functions have been designed to be re-entrant and can be used by more than one application at run time. However if two applications are to link and use the test-bed library in parallel the test-bed library can be configured as kernel module and can also be put up as a device file where the application links to the library using device function calls.

Future work should extensively involve improving the test-bed real time performance and porting the library in kernel space. RTAI offers an extensive API in utilizing its real time capabilities. Future work should involve extensive use of the RTAI API in terns of improving the functionality and performance of the test-bed library. The main problem in utilizing this test-bed is having adequate knowledge of the test-bed single board computer, test-bed library and RTAI fundamentals. The solution to this problem is to maintain adequate and easy to refer documentation with every subsequent change made to the test-bed system kernel, hardware or RTAI. To be able to refer the knowledge provided by the maintained documentation is the solution to being able to develop efficient autonomous vehicle applications.

BIBLIOGRAPHY

[1] Qing Li, Nanning Zheng, "Springrobot: A Prototype Autonomous Vehicle and Its Algorithms for Lane Detection," IEEE transactions on intelligent transportation systems, vol. 5, no. 4, December 2004.

[2] Alberto Broggi, Stefano Cattani, Pier Paolo Porta, Paolo Zani," A Laser scanner-vision Fusion System Implemented on the TerraMax Autonomous Vehicle," Proceedings of 2006 IEEE/RSJ international conference on intelligent robots and systems,Beijing,China.

[3] Daniel E. Stevenson, James D. Schwarzmeier, "Building an Autonomous Vehicle by Integrating Lego Mindstorms and a Web Cam," University of Wisconsin – Eau Claire Eau Claire WI 54702-4004.

[4] R. S. Woodley, T. Han, and L. Acar, "A new control scheme applied to the Trailer truck backing-up problem," Int. Engn. Sys.Through Art. N. Nets, vol. 6, pp.599-604, New York: ASME Press, 1996.

[5] Robert Stephen Woodley. The identification and a neural network based control for a Backward maneuvering trailer truck. Masters Thesis, University of Missouri, Rolla, 1997.

[6] R. S. Woodley and L. Acar, "Non-linear system control using path generating Neural Networks," Int. Engn. Sys. Through Art. N.Nets, vol. 8, pp. 599-604, New York: ASME Press, 1998.

[7] R. S. Woodley and L. Acar, "Neural network based control for a backward Maneuvering trailer truck," Proc.37th IEEE Conference on Decision and Control, pp. 1611-1616, IEEE Press, 1998.

[8] T. Han and L. Acar, "A neural network based approach for the identification and Optimal Control of a cantilever plate," Proc. 1997American Control Conference, v. 1, pp. 32-236, June 1997.

[9] K. Antony and L. Acar, "Real-time nonlinear optimal control using neural networks,'' Proc. 1994 American Control Conference, June-July 1994.

[10] R. S. Woodley and L. Acar, "A test-bed system for nonlinear or intelligent control," Proc. 1999 American Control Conference, pp. 3441-3445, IEEE Press, 1999.

[11] A. Macchelli, C. Melchiorri, R. Carloni, M. Guidetti. “Space Robotics: an Experimental Set-up based on RTAI-Linux,” DEIS - Dept. of Electronic, Computer Science andSystems, LAR - Lab. of Automation and Robotics, University of Bologna, Bologna, Italy.

[12] R. S. Guerra, H. J. Gon_calves J_unior, W. F. Lages. “A Digital PID Controller Using RTAI,” Electrical Engineering Department, Federal University of Rio Grande do Sul, Av. Osvaldo Aranha 103, Porto Alegre RS, CEP 90035-190, Brazil.

[13] N.Akdeniz, T. Aydın. Real BOX. “A Novel Approach to MiniRTL Implementation for Instrumentation and Control Application,” Havelsan A.Ş.06520, Ankara, Turkey.

[14] Pasi Sarolahti. “Real Time Application Interface,” Research seminar on Real-Time Linux And Java, University of Helsinki, Department of Computer Science.

[15] E Bianchi,L Dozio. “Some experiences in fast hard real time control in user space with RTAI-LXRT,” Dip.Ing. Aerospaziale, Politecnico di milano, via la masa 34, 20158 Milano, Italy.

[16] Mesa Electronics, 4I27 Users Manual, 1997.

[17] RTAI, RTAI 3.3User Manual rev0.2.

[18] Hyok-Sung Choi, Hee-Chul Yun. “Context Switching and IPC Performance Comparison Between uClinux and Linux on the ARM9 based Processor,” Software Platform Lab, Digital Media R&D Center, Samsung Electronics.

[19] Arcom Control Systems, AIM-MULTI-IO Users manual, 2007.

[20] Mesa Electronics, 4I27 Users Manual, 1997.

[21] Dimension Engineering, DE-ACCM3D 3D Users manual, 2006.

[22] Bournes electronics, 6539 precision potentiometer Users manual, 2007.

[23] Hewlett Packard Corporation, Hewlett Packard Technical Databook, 1986.

VITA

Pravin Dhake was born on February 10, 1983 in Akola, India. He received his primary and secondary education in Mumbai and graduated from RJ College, India in 2000. He graduated with a Bachelor of Engineering degree in Instrumentation Engineering from the RAIT, Mumbai University, and Mumbai, India in July 2004. He later worked as a controls engineer at Emerson Electric, Mumbai, India.

He has been enrolled in the Electrical Engineering graduate program of the University of Missouri-Rolla since August 2005, majoring in Electrical Engineering. He has been a Graduate Research Assistant in the Intelligent Systems Center from January 2006 to July 2007 and received his MS in Electrical Engineering degree in December. 2007.